Sistemas Informáticos

Curso 2006-2007

Aprendizaje Interactivo de Estructuras de Datos y Métodos Algorítmicos

Ana Isabel Saiz Jiménez

Pablo Soler Núñez

Miguel Cayeiro García

Dirigido por Clara María Segura Díaz

Dpto. Sistemas Informáticos y Programación

Facultad de Informática

Universidad Complutense de Madrid

Agradecimientos

Nos gustaría agradecer la ayuda y el apoyo que hemos recibido de las siguientes personas para el desarrollo del presente Proyecto:

- A Clara María Segura Díaz, directora de nuestro proyecto, por toda su ayuda, consejos, apoyo y tiempo dedicado, sobretodo en éstos últimos meses tan importantes para ella.
- A Rafael del Vado Vírseda, profesor de la facultad de informática, por su entusiasmo y colaboración en nuestro trabajo.
- A los profesores de Estructuras de Datos y de la Información, por permitirnos presentar nuestro proyecto a sus alumnos.
- A Javier I. Soler Areta, padre de Pablo Soler y profesor de Matemáticas en el I.S. Rafael Alberti, por su ayuda en los aspectos matemáticos del proyecto, así como por su infinita paciencia.
- Y de forma general, a nuestros padres y hermanos por todo su apoyo.

ÍNDICE

Resumen en Castellano

Este proyecto es la segunda ampliación de una herramienta pedagógica, destinada al estudio de estructuras de datos y esquemas algorítmicos, mediante visualizaciones animadas de su funcionamiento. Esta herramienta se ha estado utilizando, como complemento didáctico, durante los cursos 2005-2006 y 2006-2007, en la Facultad de Informática de la Universidad Complutense de Madrid, en las asignaturas de Estructuras de Datos y de la Información (EDI, 2º curso) y en Metodología y Tecnología de la Programación (MTP, 3º curso).

La extensión realizada sobre la herramienta ha consistido en la mejora visual, didáctica y funcional de la misma, así como en una reestructuración del código. Esta mejora se ha llevado a cabo sobre las siguientes estructuras de datos: Pilas, Colas y Árboles Binarios de Búsqueda.

Resumen en Inglés

This project is the second expansion of a pedagogical tool, destined for the study of data structure and algorithmic diagrams, using animated visualizations to explain the way it works. This tool has been used as a didactic aid in the courses of 2005-2006 and 2006-2007, in the Science Computing Department of the Universidad Complutense in Madrid, in the subjects "Data Information Structure" (Estructuras de Datos y de la Información, EDI 2nd course) and in the "Methodology and Technology of Programing" (Metodología y Tecnología de la Programación, MTP 3rd course).

The enlargement done of the tool lies on didactic, functional and visual improvement, and also a reorganization of the code. These improvements have been carried on the following data structures: Stacks, Queries and Binary Search Trees.

Palabras Clave

- **Animación:** *es la ejecución visual de una operación de una estructura de datos.*
- **Árbol binario de búsqueda:** *es un caso concreto de la estructura de datos Árbol Binario. La característica que define a este tipo de Árboles Binarios, es que los valores de los nodos están ordenados. Este orden se define de la siguiente forma: el valor contenido en todos los nodos internos es mayor o igual que los valores contenidos en su hijo izquierdo o en cualquiera de los descendientes de ese hijo, y menor o igual que los valores contenidos en su hijo derecho o en cualquiera de los descendientes de ese hijo.*
- **Cola:** *es una estructura de datos lineal y homogénea, en la que los datos entran por un extremo y salen por el otro. La cola es una estructura FIFO ya que el primer elemento de la cola será el primero en salir de ella.*
- **Estructura de datos:** *es una colección de datos cuya organización se caracteriza por las funciones definidas utilizadas para almacenar y acceder a elementos individuales de datos. Las estructuras de datos pueden descomponerse en los elementos que la forman. La manera en que se colocan los elementos dentro de la estructura afectará la forma en que se realicen los accesos a cada elemento.*
- **Operación:** *es cada una de las funciones (constructoras, modificadoras, observadoras) definidas para una estructura de datos.*
- **Pila:** *es una estructura de datos lineal y homogénea. La pila es una estructura LIFO ya que el último elemento insertado en la pila será el primero en salir de ella. El último elemento introducido en la pila se denomina cima.*
- **Simulación:** *es la secuencia completa de operaciones definida sobre una estructura de datos. Se han creado unos controles que gobiernan las simulaciones, de forma que podemos movernos a lo largo de una simulación.*
- **Visualización:** *cada una de los distintos puntos de vista bajo los cuales se muestra una estructura de datos. Las visualizaciones pueden ser: usuario (visión de la estructura de datos bajo el punto de vista de su funcionamiento externo), implementación estática usuario (visión de la estructura de datos bajo el punto de vista de su implementación estática) e implementación dinámica.*

1.- Introducción

1.1.- Antecedentes

Este proyecto, realizado en el curso 2006-2007, es la continuación de un proyecto iniciado en el curso 2004-2005 por Laura Gutiérrez García, Esther Rico Redondo y Carmen Torrano Giménez, y extendido en el curso 2005-2006 por Eduardo de la Iglesia Ramírez, Gonzalo Moreno Pereira y Cristina Rubert Sánchez.

La motivación principal del trabajo que se ha realizado durante estos 3 años ha sido, básicamente, desarrollar una herramienta útil que permitiera:

- En primer lugar, a los profesores de las asignaturas de EDI (Estructura de Datos y de la Información) y MTP (Metodología y Tecnología de la Programación), explicar el contenido de sus asignaturas con la ayuda de una aplicación interactiva.
- En segundo lugar, dar todas las facilidades a los alumnos de dichas asignaturas para comprender, tanto el comportamiento como las distintas formas de implementar estructuras de datos, así como el funcionamiento interno de distintos métodos algorítmicos.

A esta aplicación la llamamos **Visualización de Estructuras de Datos y Algoritmos**, en lo sucesivo **VEDYA**.

Durante el primer año de desarrollo del proyecto (curso 2004-2005) se llevó a cabo el diseño y la implementación, en JAVA, de una aplicación que permitía la visualización de cinco estructuras de datos, a saber, pilas, colas, arboles binarios de búsqueda, arboles AVL y colas de prioridad, así como varios algoritmos que aplican los esquemas algorítmicos siguientes: algoritmos voraces, programación dinámica, divide y vencerás, y vuelta atrás.

A lo largo del segundo año, se realizaron mejoras en la implementación, como la modularización de distintas clases, y se continuó ampliando el elenco de estructuras de datos disponibles, con dos nuevas, tablas ordenadas y tablas hash, así como la

implementación de un sistema de gestión de un consultorio médico donde los pacientes pueden ser atendidos por varios médicos.

1.2.- PROPÓSITO

La utilización de la aplicación **VEDYA** durante el curso 2005-2006 en la asignatura de Estructura de Datos y de la Información (EDI), nos permitió conocer la existencia de este proyecto y nos motivó para aportar ideas nuevas en el desarrollo de la herramienta. Durante el uso de **VEDYA** comprobamos que aunque el funcionamiento era correcto, en la visualización de ciertas animaciones no quedaban suficientemente claros todos los pasos de las operaciones que se realizaban y pensamos se le podría mejorar ciertos aspectos de la aplicación.

Una vez se nos fue concedido el proyecto **VEDYA** para desarrollarlo dentro de la asignatura de Sistemas Informáticos del curso 2006-2007, nos fijamos los objetivos que queríamos conseguir en la realización del proyecto, siendo principalmente los siguientes:

- Mejorar la interfaz gráfica utilizando instrumentos gráficos más potentes de forma que las animaciones resultasen más fluidas, atractivas y que explicasen mejor el funcionamiento interno de las operaciones realizadas.
- Inclusión de objetos de aprendizaje como parte de la herramienta, para que guiara el aprendizaje de los alumnos. Dichos objetos estarían basados en documentación de las asignaturas correspondientes.
- Ampliación de la funcionalidad de la herramienta, centrándonos principalmente en la parte relacionada con los esquemas algorítmicos. En particular queríamos incorporar algoritmos probabilistas.

1.3.- MODIFICACIÓN DE LA HERRAMIENTA

A priori cabe pensar, que el trabajo realizado durante el desarrollo de este proyecto, ha sido una simple extensión de **VEDYA**. Pero nada más lejos de la realidad, debido a que no nos hemos conformado con lo que había, y hemos querido añadirle una funcionalidad más potente a la aplicación.

Debido a la transformación tan radical a la que hemos sometido a **VEDYA**, sobre todo en el aspecto gráfico, consideramos que la mejor forma de realizar los cambios que pretendíamos hacer a la aplicación era la reimplementación completa del código. No obstante, hemos mantenido la estructura básica del diseño así como muchas de las ideas utilizadas en las versiones anteriores de **VEDYA**.

Debido a que las versiones anteriores de **VEDYA** estaban implementadas en **JAVA**, y siendo este lenguaje muy extendido, hemos decidido mantenerlo.

1.4.- ¿Por qué?

Las razones principales por las que se ha llevado a cabo esta profunda modificación de la aplicación, tienen que ver, fundamentalmente, con diversas carencias encontradas al utilizarla. Al evaluar estas carencias las agrupamos de la siguiente forma:

1.4.1.- Deficiencias gráficas

A pesar del esfuerzo dedicado en el desarrollo de **VEDYA** por parte de nuestros predecesores, nos pareció que se podía mejorar el interfaz gráfico. Para ello se ha intentado diseñar una interfaz con un aspecto más moderno y amigable de manera que sea atractivo utilizar **VEDYA**.

Para ver el cambio de aspecto al que se ha sometido la aplicación, en la ***figura* 1** se muestra como era **VEDYA** en versiones anteriores, y en la ***figura* 2** se muestra el nuevo interfaz.

Figura 1 - Versión anterior de **VEDYA**

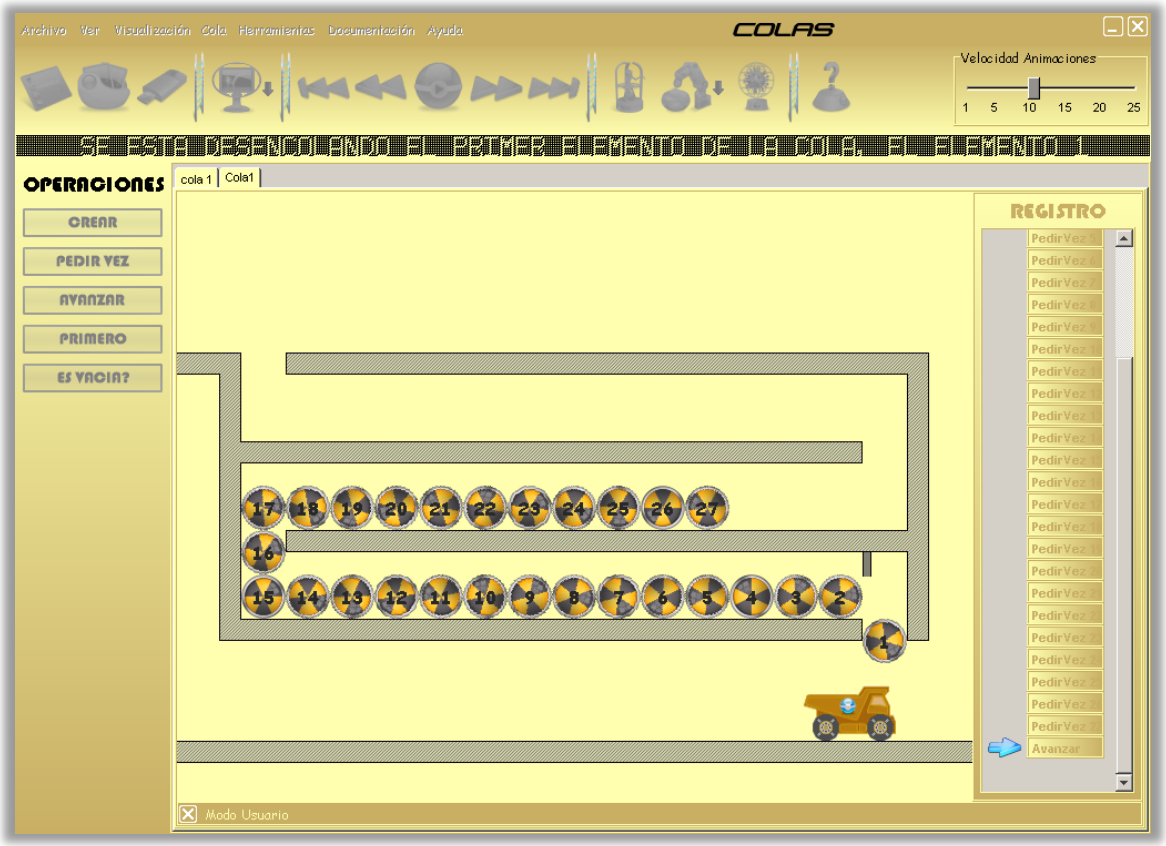


Figura 2 - Versión actual de **VEDYA**

1.4.2.- Deficiencias funcionales

La funcionalidad de la que disfrutaba **VEDYA** era bastante reducida, puesto que las opciones de manejo de la aplicación eran:

- Realizar operaciones con la estructura de datos que se ha seleccionado.
- Para poder ver mejor las operaciones que se han realizado, en una parte de la pantalla se muestra el estado actual de la estructura, mientras que en otra parte se muestra el estado anterior.
- Para una mejor comprensión de las estructuras de datos, se puede visualizar bajo distintos puntos de vista:
  - Una vista general de la estructura que intenta introducir al usuario en su significado.
  - Distintas visualizaciones que representan las distintas implementaciones de la estructura de datos.
- Cargar una prueba predeterminada por los programadores.
- Consultar tanto la documentación individualizada de cada estructura de datos así como la ayuda de la aplicación.

A esta nueva versión de **VEDYA** se le ha añadido una mayor variedad en las opciones disponibles, pudiendo realizar casi cualquier acción útil para llevar a cabo el objetivo para el cual está destinada esta aplicación, esto es, para una mejor comprensión y aprendizaje de diversas estructuras de datos. Dentro de las nuevas características que se le han añadido, cabe destacar las siguientes:

- Ejecución en diferentes Sistemas Operativos (Windows, Linux, Mac).
- Subsistema de carga y almacenamiento de estructuras de datos (ver **secciones 3.4.1, 3.4.7.2** y **6.1**). Esta nueva característica nos permite almacenar una estructura de datos en un fichero como una secuencia de operaciones válida. Así como su posterior carga dentro de la aplicación, asegurándonos que el archivo sea realmente una secuencia de operaciones válida.
- Subsistema multiventana y multipestaña (ver **sección 4.2**). Se ha creído conveniente añadir la posibilidad de tener abiertas todas las estructuras de datos que queramos. Para permitir esta libertad, hemos estructurado el interfaz de la siguiente forma.

- Los contenedores de las estructuras de datos son pestañas.
- Los contenedores de las pestañas son ventanas.
- Cada ventana está especializada en un tipo concreto de estructura de datos, por lo que cada ventana sólo contendrá pestañas que contengan las estructuras de datos a las que hace referencia.

Por ejemplo, se puede crear una ventana de Pilas, así como una ventana de Colas. Pues bien, la ventana de Pilas sólo contendrá pestañas que contengan pilas y la ventana de Colas sólo contendrá pestañas que contengan colas.

En la ***figura 3*** se puede observar que se pueden tener varias ventanas abiertas.

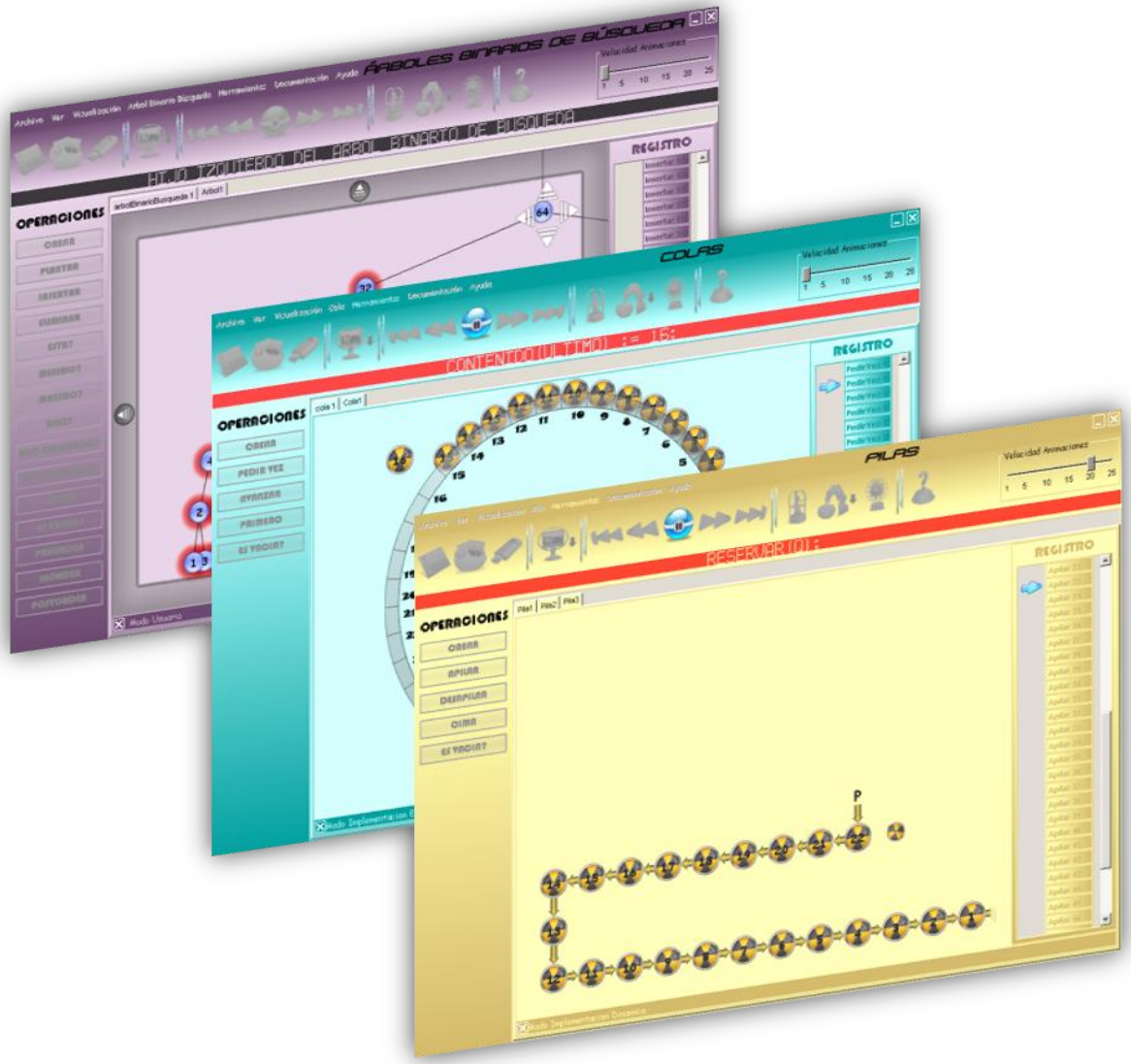


Figura 3 - Sistema multiventana y multipestaña

En este ejemplo tenemos, en concreto, una ventana donde se están simulando pilas, otra donde se están simulando colas y en la otra, árboles binarios de búsqueda. Además en cada una de las ventanas hay varias pestañas donde se están ejecutando distintas secuencias de operaciones de las estructuras.

- Subsistema de control de la simulación (ver **secciones 3.4.5** y **6.1**). Se nos brinda la opción de manipular la simulación de la estructura a nuestro antojo, dándonos la posibilidad de avanzar o retroceder a cualquier operación realizada hasta el momento. De esta forma podremos repetir operaciones que no han quedado claras, o construir nuevas simulaciones a partir de cualquier punto de una simulación ya existente, sin necesidad de tener que repetir el proceso de creación desde el principio, como en la herramienta anterior.
- Se nos permite modificar la velocidad a la que se van a ejecutar las operaciones. Esto nos da la posibilidad de examinar con más detalle las operaciones más complejas.
- Las estructuras más complejas no tienen límite de tamaño. Mediante unos controles especiales (ver **secciones 3.4.7.1** y **6.3**) se permite, cambiar el tamaño y la posición de la estructura de datos dentro de la pestaña.

1.4.3.- Deficiencias didácticas

Uno de los mayores problemas que le encontramos a **VEDYA** al trabajar con ella, fue la dificultad que suponía comprender las operaciones más complejas. Las animaciones que se habían implementado para reflejar cada una de las acciones que se podían realizar en una estructura de datos, no resultaban todo lo claras y explicativas que cabía esperar.

En esta nueva versión de **VEDYA**, se ha hecho especial hincapié en que las animaciones reflejen fielmente cada una de las instrucciones que se realizan al ejecutar una operación de una estructura de datos. Para ello, en cada una de las visualizaciones que muestran las distintas implementaciones que puede tener una estructura de datos, se muestra el pseudocódigo de cada operación paso a paso, a la vez que se visualiza la animación correspondiente (ver **sección 3.3**).

Para ver las mejoras que se han logrado a lo largo de este año en este aspecto, en las **figuras 4** y **5**, se muestra la operación eliminar sobre un mismo árbol, en la versión anterior de **VEDYA** y en la nueva, respectivamente. Para mostrar este ejemplo, se ha elegido la eliminación del elemento 50 que tiene hijo izquierdo e hijo derecho.

En la **figura 4**, en el segundo y tercer paso, se señala que en el lugar del elemento 50 se va a colocar el elemento 70 y se tacha el primero. A continuación la simulación de la operación muestra como el elemento se mueve horizontalmente hacia la derecha hasta desaparecer.

En la **figura 5**, se muestran todos los pasos que se realizan hasta eliminar el elemento 50. Así, primero busca donde está el elemento, una vez que lo ha encontrado y como tiene los dos hijos, se observa como busca el elemento mínimo de su hijo derecho. Una vez encontrado, se intercambian ambos y, como ahora el elemento 50 solo tiene un hijo, se ve como se elimina poniendo su hijo derecho en su lugar.

A parte, se ha desarrollado un subsistema de aprendizaje que consiste en evaluar mediante tests los conocimientos adquiridos, por parte del alumno, de cada una de las estructuras de datos. Este subsistema está formado por dos partes:

- Se ha creado una nueva aplicación, **VEDYA-TEST**, (ver **secciones 3.5** y **5**) destinada al uso exclusivo por parte del profesor. Esta aplicación permite el almacenamiento de preguntas en una base de datos, con las que se pueden crear tests.
- En **VEDYA** se ha añadido la posibilidad de abrir tests (ver **sección 3.4.6**), creados con la herramienta **VEDYA-TEST**, de tal forma que el alumno pueda responder las preguntas del test con la posibilidad de que **VEDYA** lo autocorrija.

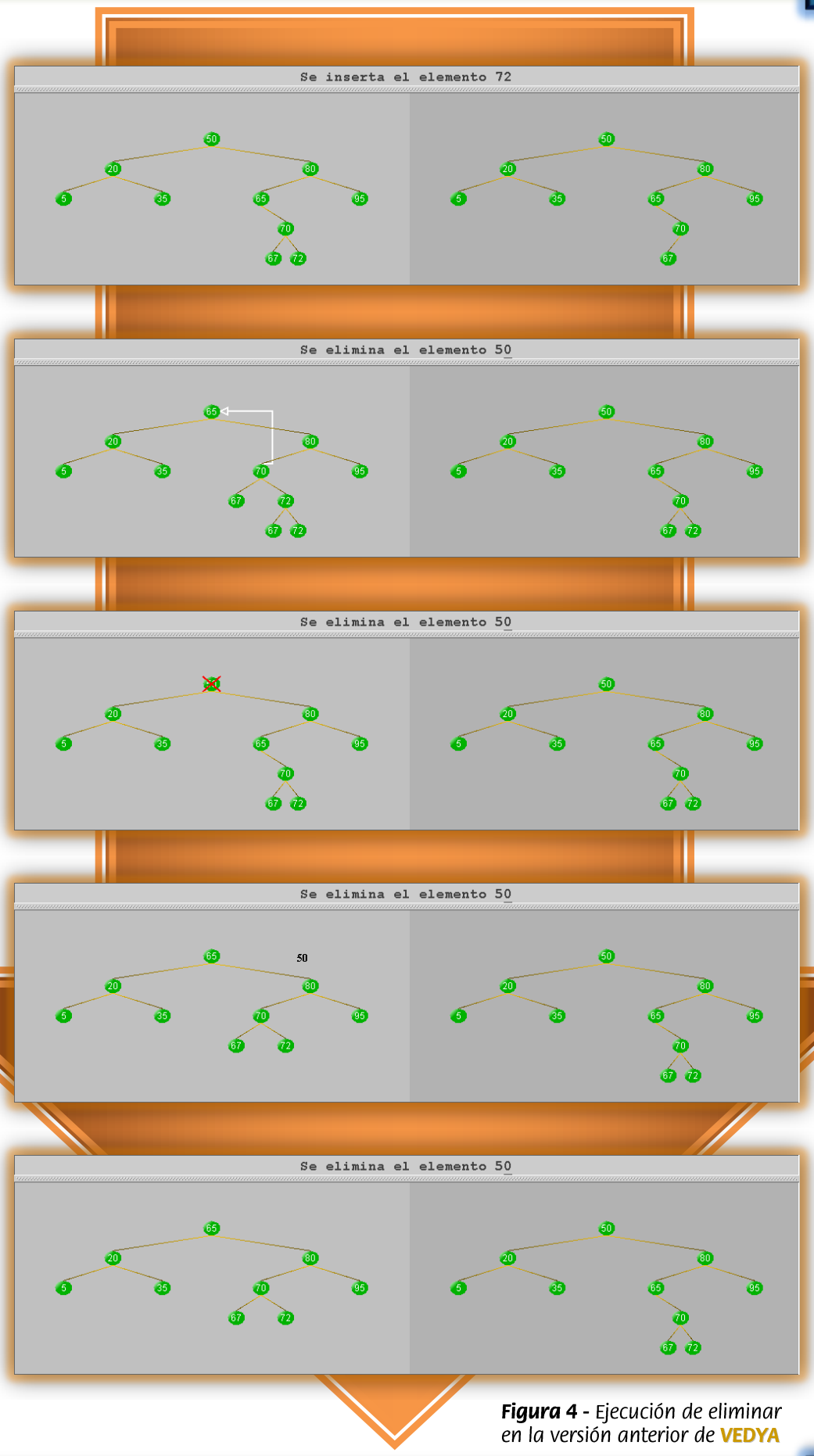
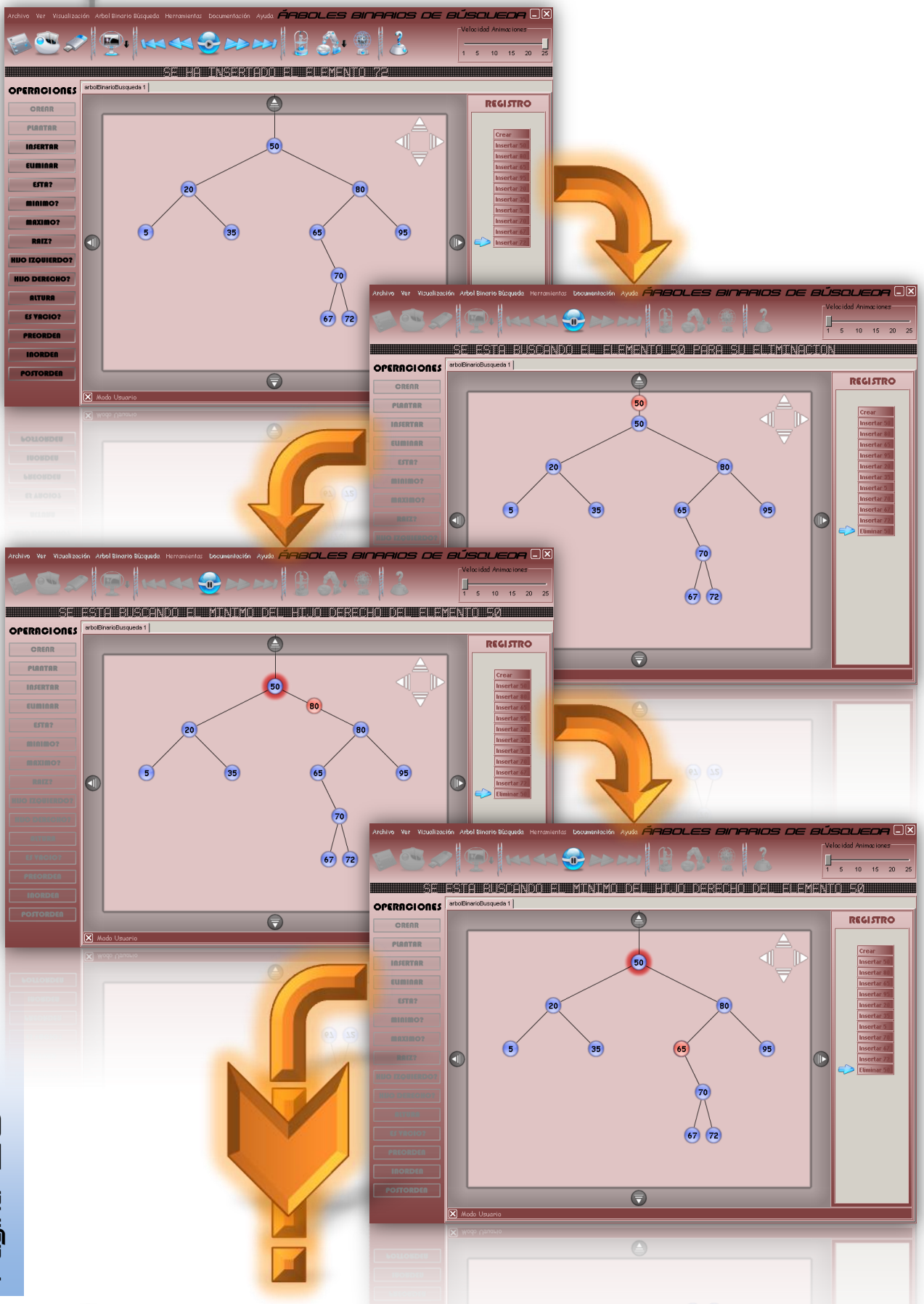


Figura 4 - Ejecución de eliminar en la versión anterior de **VEDYA**


Árboles Binarios de Búsqueda
Se ha insertado el elemento 72
Se está buscando el elemento 50 para su eliminación
Se está buscando el mínimo del hijo derecho del elemento 50
Se está buscando el mínimo del hijo derecho del elemento 50
Operaciones
Registro
Crear
Insertar 50
Insertar 80
Insertar 65
Insertar 95
Insertar 20
Insertar 5
Insertar 35
Insertar 70
Insertar 67
Insertar 72
Eliminar 50
Modo Usuario

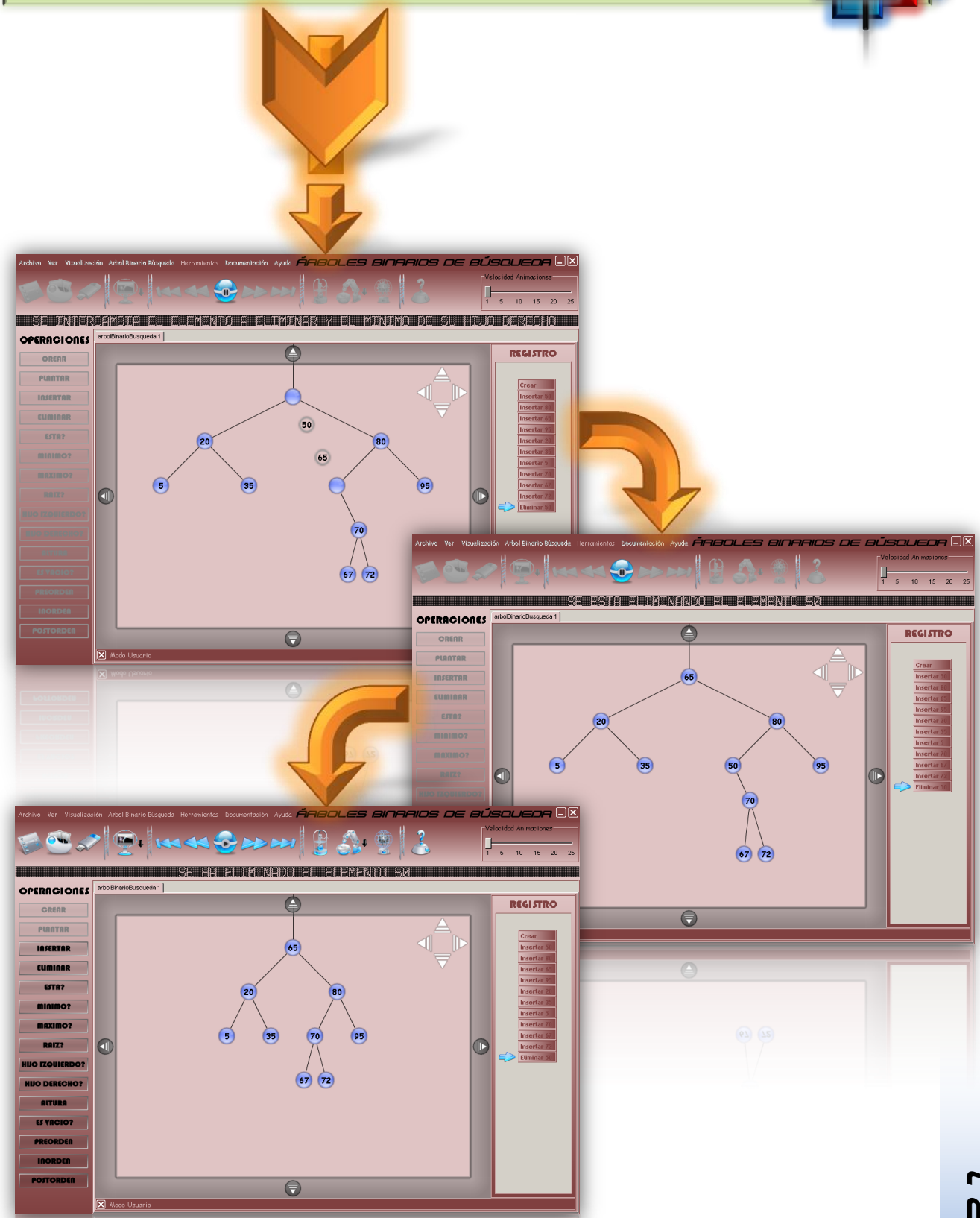


Figura 5 - Ejecución de eliminar en la versión actual de **VEDYA**

1.4.4.- Deficiencias en la implementación

Al empezar a comprender el código para trabajar sobre el proyecto, nos percatamos que había una serie de problemas en la implementación.

- Falta de modularidad del código.
- Poca reutilización de código.
- Implementación de ciertos métodos de forma ineficientes.

Aprovechando que se ha reimplementado la aplicación por completo hemos intentado subsanar estos problemas (ver **secciones 4.1** y **4.2**). Para ello:

- Se ha intentado que cada una de las clases que se han creado sean lo más específicas posibles para que no resulten muy largas y complicadas de entender. Como resultado hemos obtenido clases que, por lo general, no superan las 1000 líneas.
- Se han intentado utilizar al máximo dos de las características más potentes de los lenguajes orientados a objetos, como son la herencia y el polimorfismo.
- Se ha tenido especial cuidado en que la implementación de los métodos sea lo más eficiente posible.

1.5.- ¿Cómo?

Para añadir todas estas nuevas características, se ha rediseñado la estructura de la herramienta, de tal forma que aunque la idea básica del diseño es la misma, han cambiado algunos conceptos.

Lo que hay que tener claro en **VEDYA**, es que para dibujar una estructura de datos y las animaciones que se realizan sobre ella, se utiliza la implementación de la propia estructura de datos.

1.5.1.- Diseño anterior

En el primer diseño de la herramienta, se trabajó mucho para conseguir que fuese lo más modular posible con respecto a cada una de las partes fundamentales de la aplicación. Para ello se realizaron tres módulos principales, la interfaz, el bloque de

gráficos y el bloque de implementaciones, de tal forma que las implementaciones no se mezclasen nunca con la parte gráfica.

En la **figura 6** se representa un esquema del diseño que se utilizó para desarrollar la herramienta. A continuación se explican sus características principales.

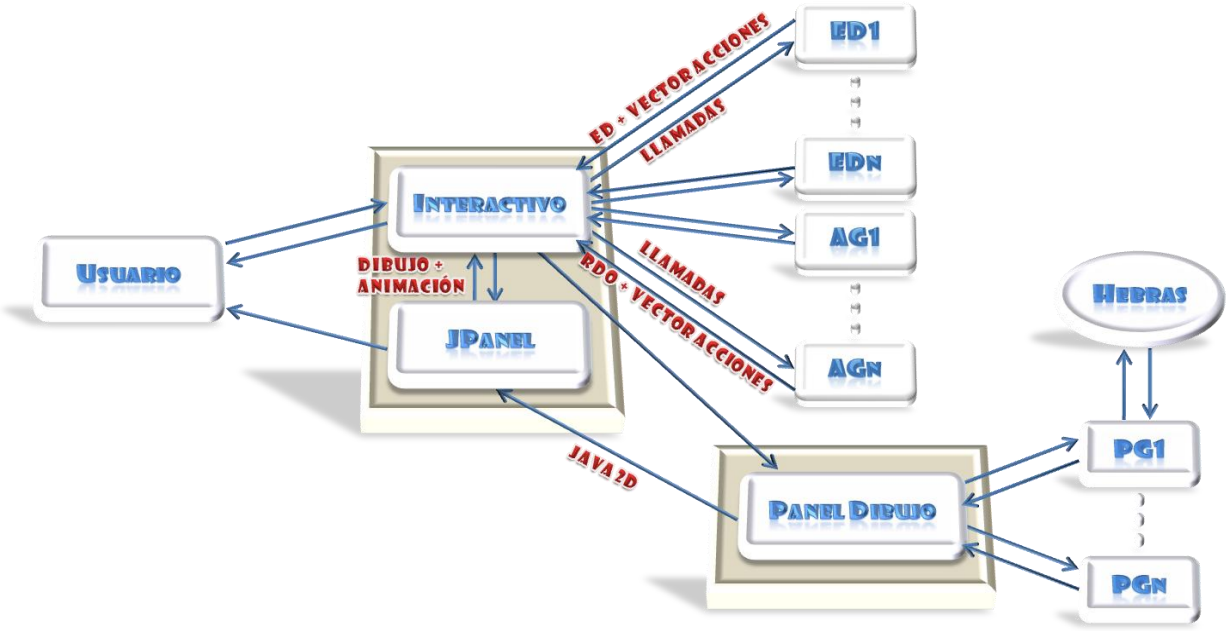


Figura 6 - Diseño anterior de **VEDYA**

El usuario se comunica con la herramienta a través de la interfaz, la cual posee una parte interactiva y una parte a la que se denominó JPanel, donde se puede visualizar la animación de las distintas estructuras de datos y algoritmos. La parte de visualización muestra los efectos de la interacción del usuario con la parte interactiva mediante dibujos y animaciones.

La parte interactiva, en función de las operaciones aplicadas por el usuario, llama a la parte de implementación de las estructuras de datos o de los algoritmos, las cuales devolverán:

- En el caso de trabajar con una estructura de datos: la estructura de datos modificada y un vector de acciones que contiene las acciones que se tienen que realizar para representar gráficamente una operación. Estas acciones recogen la información que necesita la parte gráfica.

- En el caso de trabajar con un algoritmo, el resultado de la ejecución del problema con el que se está trabajando y un vector de acciones para la representación gráfica posterior.

A su vez la parte interactiva llama al panel de dibujo, el cual actúa a modo de fachada, es decir, se encarga de la comunicación con la parte gráfica, de tal forma que cuando la interfaz desee pintar, deberá pedírselo al panel de dibujo, el cual mostrará los resultados obtenidos de su llamada a la correspondiente parte gráfica, sobre el JPanel.

En la parte gráfica se han desarrollado tanto gráficos, como animaciones de las distintas estructuras de datos y algoritmos. El movimiento de las animaciones que tienen lugar en la parte gráfica, son controladas mediante hebras.

Este diseño supone que cada parte es responsable de una tarea. De tal forma, la parte gráfica sólo dibujará y animará, y la parte de implementación únicamente efectuará las operaciones de la correspondiente estructura de datos o resolverá un determinado problema algorítmico. Esto implica que la parte de implementación no sabe dibujar y que la parte gráfica no sabe cómo está implementada la estructura de datos o algoritmo.

1.5.2.- Diseño actual

En esta nueva versión de **VEDYA**, se ha mantenido la estructura básica del diseño original, ya que hemos considerado que se realizó un buen trabajo y no tenía sentido cambiarlo. Las modificaciones que hemos introducido, a nivel de diseño, han ido dirigidas a aumentar las posibilidades de la aplicación.

En la ***figura 7*** se representa un esquema del nuevo diseño de **VEDYA**. Como se puede comprobar, guarda muchas semejanzas con el diseño original, aunque se han modificado convenientemente ciertos aspectos que a continuación detallamos.

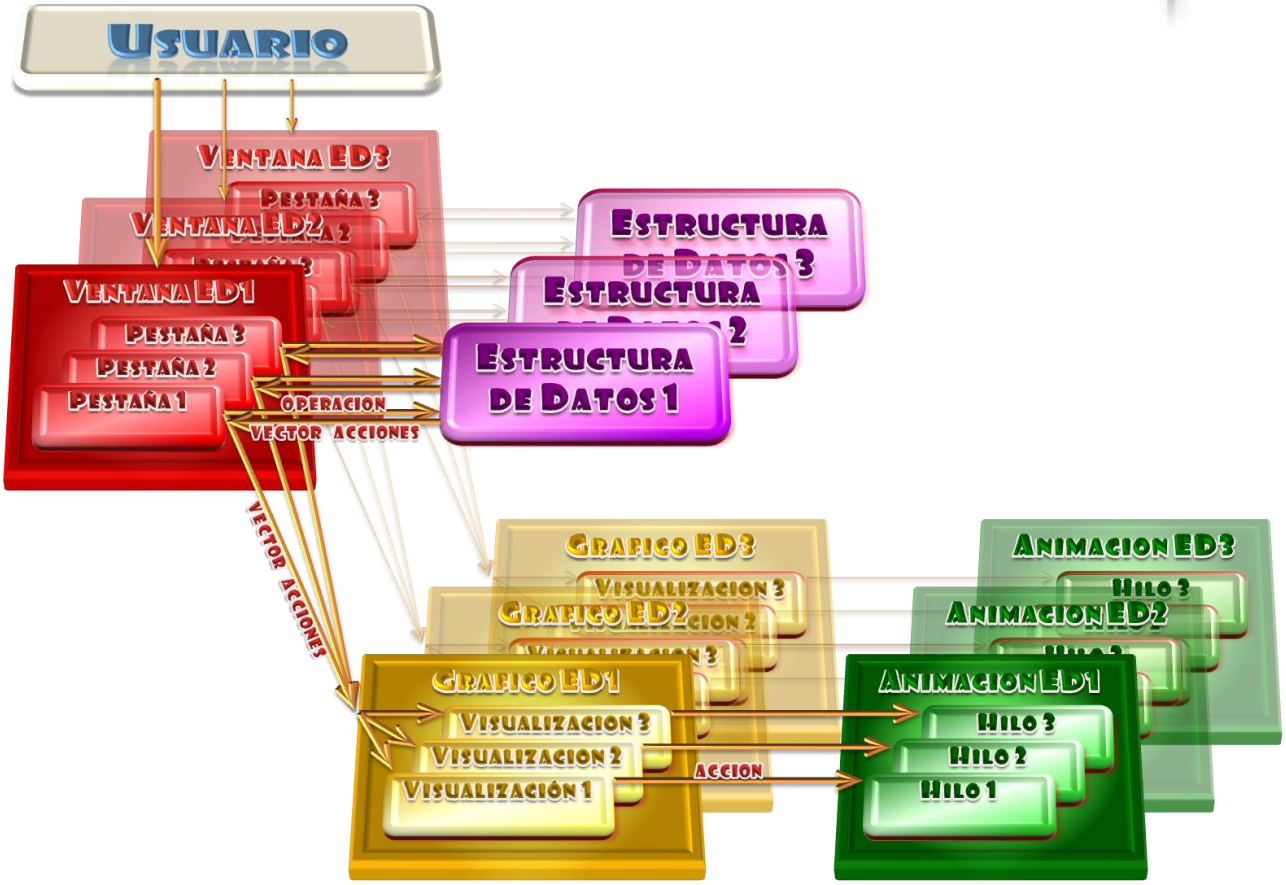


Figura 7 - Diseño actual de **VEDYA**

En este diseño se ha querido reflejar la modularización que hemos realizado en este proyecto, no solo a nivel de bloques, sino también la que se ha llevado a cabo, a grandes rasgos, dentro de cada uno de estos.

En primer lugar, se han añadido opciones multiventana y multipestaña que permiten la coexistencia simultánea de una ventana por cada estructura, así como la coexistencia dentro de cada ventana, de todas las pestañas (por cada pestaña, una única visualización) que se deseen con las estructuras de datos. Esto implica que el usuario puede manejar simultáneamente distintas ventanas y distintas pestañas.

A su vez cada pestaña puede tener varios tipos de visualizaciones, que dibujan la estructura de datos acorde a lo que quiere representar, ejecutando cada una de estas visualizaciones, el hilo correspondiente, para mostrar las animaciones.

Además, también se ha modificado la gestión de la parte gráfica. Esta modificación ha consistido en la modularización de la estructura de datos gráfica en cada una de las distintas visualizaciones que posea dicha estructura de datos.

Por otra parte, aunque se ha mantenido la idea general del funcionamiento interno de la aplicación, se han redefinido las acciones utilizadas para mostrar las animaciones correspondientes (ver **secciones 6.4** y **6.6**). Esta redefinición ha sido necesaria ya que al cambiar la forma de representar gráficamente las estructuras de datos, requeríamos una información más precisa por parte de la implementación.

El funcionamiento básico de la aplicación se detalla en la **figura 8**.

Figura 8 – Secuencia de pasos para la ejecución de una operación

El usuario tiene abierta una ventana de una cierta estructura de datos, con al menos una pestaña, y ejecuta una de las múltiples operaciones que puede realizar. La ventana procesa la operación pulsada, y la envía a la pestaña que está seleccionada. La pestaña procesa la operación y la envía a la estructura de datos. Se ejecuta la operación sobre la estructura de datos, y ésta, devuelve a la pestaña, el vector de acciones que se ha generado al realizar la operación. La pestaña envía el vector de acciones a la estructura de datos gráfica seleccionada, que gestiona el procesado del vector de acciones correspondiente a la operación. La estructura de datos gráfica prepara los gráficos necesarios para cada acción y envía dicha acción al hilo que la ejecuta mostrando su animación.

1.6.- Alcance de la Aplicación

El eje central del desarrollo de este proyecto ha sido la modificación de la interfaz gráfica. A la vez que se ha mejorado estéticamente, también se ha mejorado la interacción con el usuario, de tal manera que las posibilidades en la creación de nuevas visualizaciones de estructuras de datos, son realmente grandes:

- Se ha conseguido el funcionamiento de la aplicación bajo distintos Sistemas Operativos, en concreto, Windows, Linux y Mac.
- El diseño de la aplicación ha sido pensado para un uso fácil e intuitivo por parte del usuario.
- A la ventana principal de cada estructura se le ha dotado de una gran funcionalidad.
  - A parte de poder crear una simulación de una estructura de datos ejecutando sus operaciones, el usuario puede moverse a lo largo de la secuencia de operaciones de tal forma que se pueda repetir la simulación o incluso modificar la simulación a partir de cualquier operación.
  - Posibilidad de cambiar la velocidad de las animaciones de manera que su visualización sea más sencilla.
  - Guardar la simulación de una estructura de datos definida por sus operaciones, brindando la posibilidad de cargar posteriormente esa misma simulación.
  - Se pueden tener abiertas simultáneamente todas las simulaciones de estructuras de datos que se quiera, pudiendo cambiar de una a otra para que una fácil comparación entre ellas.
- Las representaciones gráficas de cada una de las estructuras son originales, en cuanto que es una visión que no se encuentra en los libros. Se ha buscado que tengan un efecto didáctico, ya que la filosofía propia de cada estructura se ha tenido muy en cuenta.
- Las visualizaciones de cada estructura están capacitadas para contener una cantidad relevante de elementos. Así, el usuario, tiene la posibilidad de ver el comportamiento de estructuras grandes.

- En las **figuras 9**, **10** y **11** se muestra la estructura de datos pila en sus tres visualizaciones con el número máximo de elementos que hemos considerado para esta estructura, es decir 52 elementos.
- En las **figuras 12**, **13** y **14** se muestra la estructura de datos cola en sus tres visualizaciones con el número máximo de elementos que hemos considerado para esta estructura, es decir 40 elementos.
- En la **figura 15** se muestra un árbol binario de búsqueda de 128 elementos en 7 niveles. Como para esta estructura no ponemos límite en el número de elementos, hemos añadido controles de tamaño y posición de la estructura, que nos permiten configurar la imagen para que quepan un gran número de elementos en la pantalla.

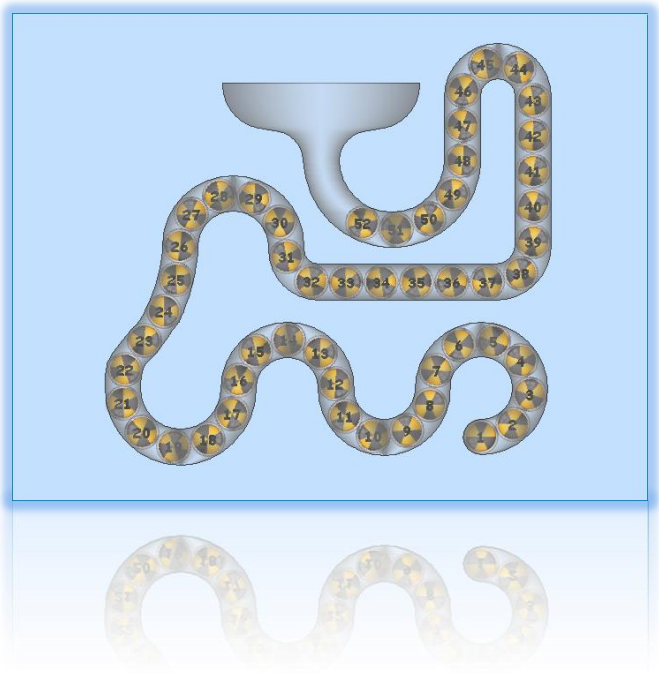


Figura 9 – Pila llena. Visualización modo Usuario

Figura 10 – Pila llena. Visualización modo Implementación Estática

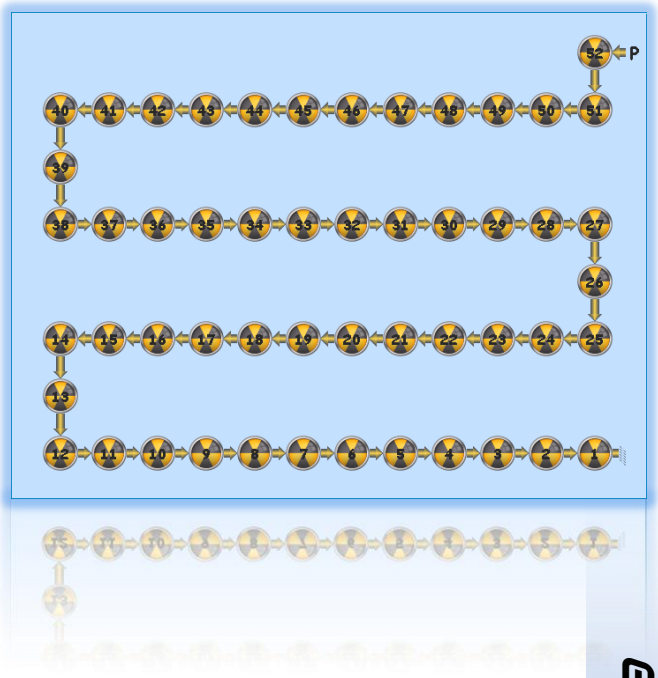


Figura 11 – Pila llena. Visualización modo Implementación Dinámica

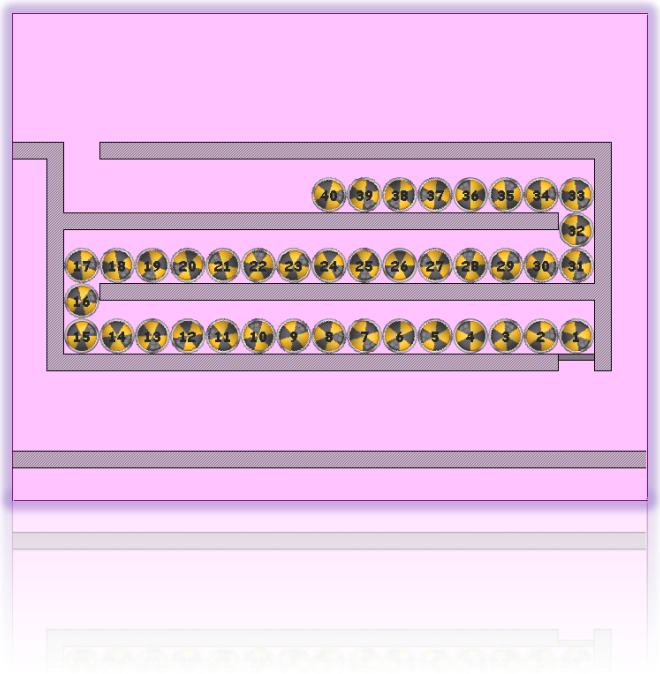


Figura 12 – *Cola llena. Visualización modo Usuario*

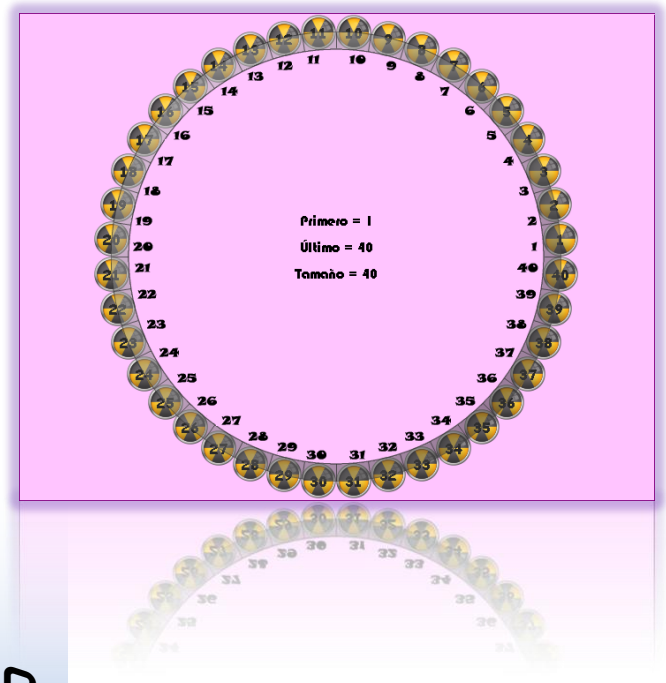


Figura 13 – *Cola llena. Visualización modo Implementación Estática*

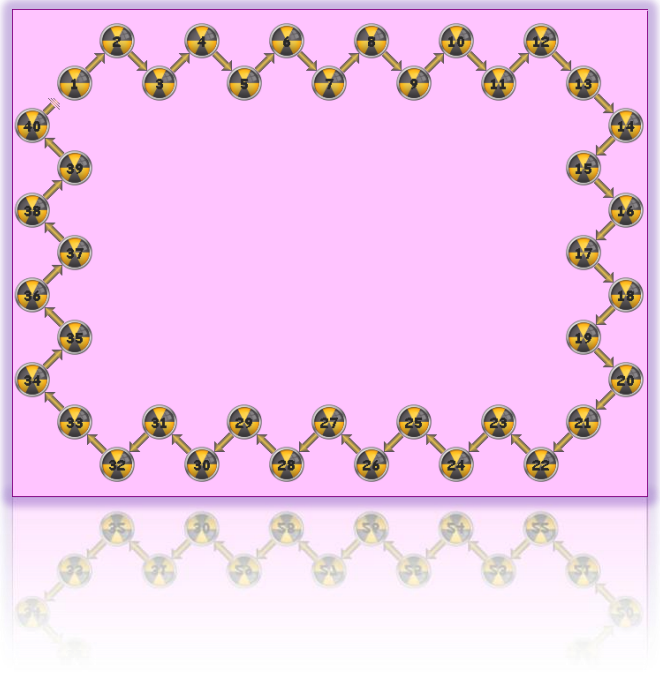


Figura 14 – *Cola llena. Visualización modo Implementación Dinámica*

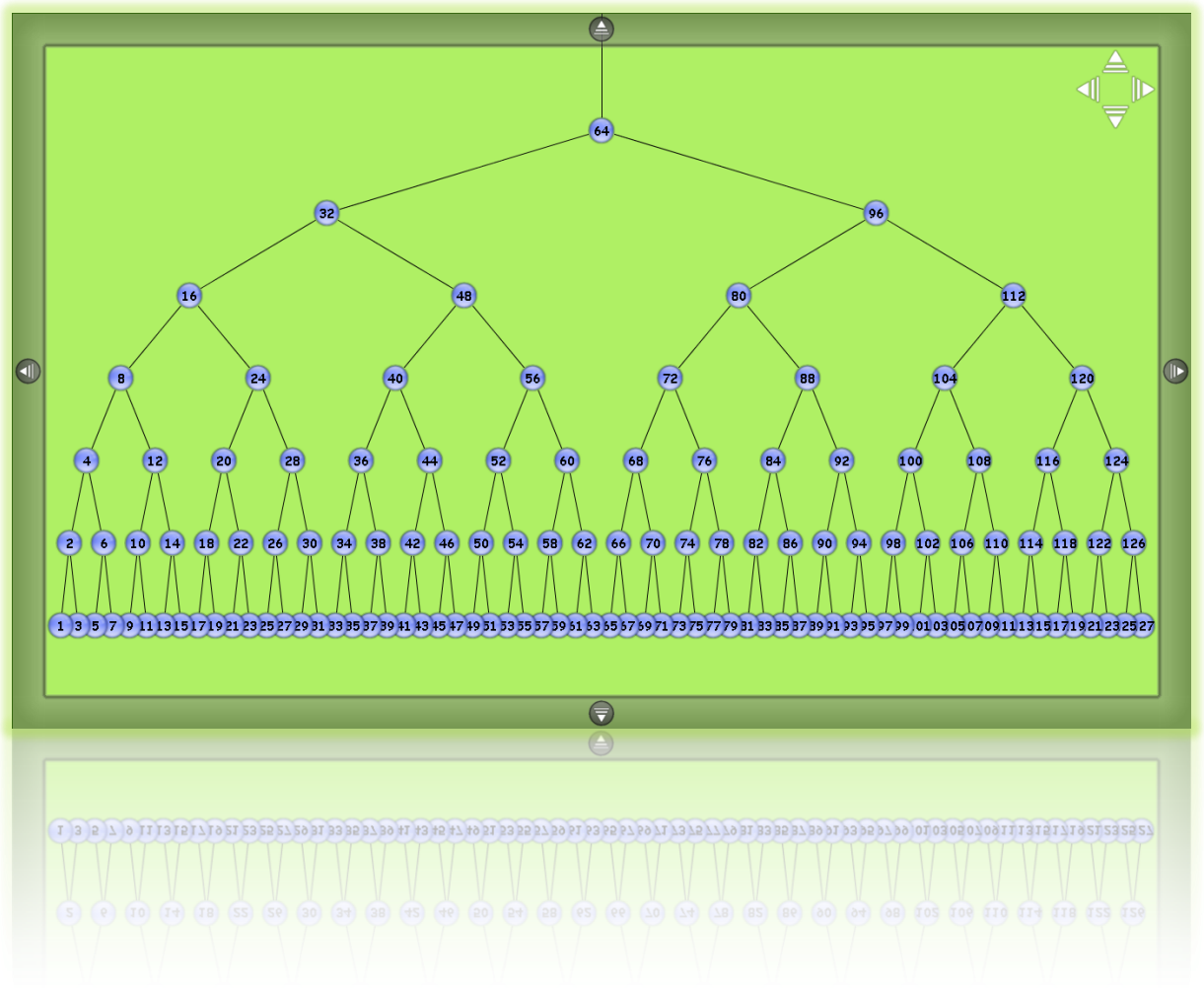


Figura 15 – Árbol Binario de Búsqueda de 7 niveles. Visualización modo Usuario

- Se han logrado implementar unas animaciones muy vistosas y realistas de las operaciones. Las animaciones de cada operación se han desglosado en acciones, a las que hemos definido como las unidades mínimas de animación, de tal forma que se muestran con todo detalle cada uno de los pasos por los que está compuesta una operación.
- Aunque solamente se ha conseguido implementar las visualizaciones de tres estructuras de datos, la ampliación de la aplicación con nuevas estructuras, es realmente sencillo gracias al buen diseño e implementación realizados.
- Se le ha añadido a **VEDYA** una parte completamente nueva, útil en la interacción profesor- alumno, que consiste en realización de tests, creados por el profesor con la también nueva aplicación **VEDYA-TEST**, para su posterior realización por parte de los alumnos.

1.7.- Visión General del Documento

Este documento se compone de ocho secciones, se pretende explicar la aplicación partiendo desde una visión general, para ir profundizando en la explicación con un mayor nivel de detalle:

- En la sección **"2.-Perfil de usuario"**: se describen las características de los usuarios, para los cuáles se ha diseñado la aplicación.
- En la sección **"3.-Descripción general de la herramienta"**: se ofrece una descripción pormenorizada del funcionamiento general del sistema, explicando con detalle cada una de las estructuras de datos implementadas. Se muestran diferentes ejemplos de utilización de cada una de las estructuras de datos, en todas y cada una de sus visualizaciones.
- En la sección **"4.-Estructura de la aplicación VEDYA"**: se da una explicación de la estructura de **VEDYA**, mostrando sus diagramas de paquetes y de clases. Exponemos los cambios realizados sobre la estructura de los años anteriores, razonando los motivos por los cuales se ha decidido efectuarlos. Comentamos que se ha continuado con la idea de modularizar el proyecto, y realizamos una breve descripción de cada una de las clases, organizadas por paquetes, útil para tener una visión global del funcionamiento interno de **VEDYA**.
- En la sección **"5.-Estructura de la aplicación VEDYA-TEST"**: se explica la estructura de **VEDYA-TEST**, mostrando sus diagramas de paquetes y de clases. Argumentamos los motivos de esta estructura, al igual que realizamos una breve descripción de cada una de las clases, organizadas por paquetes, útil para tener una visión global del funcionamiento interno de la **VEDYA-TEST**.
- En la sección **"6.-Aspectos técnicos de la implementación"**: se describen con mayor detalle ciertos aspectos técnicos que hemos creído conveniente destacar:
  - Cómo realizamos la carga y almacenamiento de las estructuras.
  - Cuáles son los controles de simulación y cómo es su funcionamiento, incluyendo tres diagramas de secuencia que muestran cada una de las llamadas a funciones que se realizan al utilizar alguno de de estos controles.
  - Mejora de la aplicación con respecto al tamaño de las estructuras, explicando las técnicas utilizadas para conseguir estructuras de gran

tamaño. Análisis del funcionamiento de los controles de tamaño y posición.

- Descripción detallada de cada una de las acciones que componen una operación de una estructura de datos. Razones por las que se han diseñado de esa forma. Explicación de la importancia del vector de acciones y como interactúa en cada una de las clases en las que es utilizado.
- Explicación del cambio en la forma de dibujar los elementos respecto a los años anteriores, así como las mejoras que hemos conseguido gracias a ello.
- Descripción de los aspectos técnicos más importantes, utilizados en las implementaciones de cada una de las visualizaciones de las estructuras de datos. Se explican los diseños de cada una de las estructuras, así como el movimiento de los elementos en las simulaciones, tales como giros, desplazamientos, etc.

- En la sección **"7.-Cómo ampliar VEDYA"**: se expone una explicación detallada, organizada por paquetes, de cada uno de los pasos a seguir para añadir una nueva estructura. Se especifica qué clases se tienen que crear y donde, así como las clases que se tienen que modificar y cómo.
- En la sección **"8.-Trabajo futuro"**: se exponen los objetivos propuestos para la ampliación de la aplicación, durante el desarrollo de la beca concedida a uno de los componentes del grupo.
- En la sección **"9.-Valoración del trabajo realizado"**: comentamos las destrezas adquiridas a lo largo de la realización del proyecto, junto a una valoración personal del trabajo realizado. Además se comenta la experiencia conseguida mediante exposiciones de la aplicación.

2.- PERFIL DE USUARIO

Esta herramienta está dirigida, fundamentalmente, a alumnos de informática que cursen las asignaturas de Estructuras de Datos y de la Información y Metodología y Tecnología de la Programación. Esta aplicación también resultará útil a los profesores que imparten dichas asignaturas. En la nueva versión de **VEDYA**, debido a que se está reimplementando de nuevo el código, hasta el momento solamente se han podido añadir estructuras de datos, por lo que para la parte de los algoritmos se tendrá que seguir usando la versión anterior de **VEDYA**.

En Ingeniería Informática e Ingeniería Técnica en Informática de Gestión EDI consta de 12 créditos y MTP de 15. En Ingeniería Técnica en Informática de Sistemas EDI cuenta con 12 créditos y MTP con 12 créditos. Los descriptores oficiales del plan de estudios de la Facultad de Informática de la Universidad Complutense de Madrid son:

EDI	MTP
Diseño de algoritmos recursivos.	Diseño de algoritmos.
Tipos abstractos de datos: especificación e implementación.	Análisis de algoritmos.
Estructuras de datos y algoritmos de manipulación.	Lenguajes de Programación.
Estructuras de la Información: ficheros, bases de datos.	Diseño de programas: descomposición modular.
	Técnicas de verificación y prueba de programas.

Tabla 1 – Temario de EDI y MTP

Los conocimientos recomendados para las personas que usen esta herramienta son:

- Temas de verificación de programas y descomposición modular.
- Conocimientos básicos de un lenguaje imperativo (instrucciones, tipos de datos, procedimientos).
- Diseño descendente de programas.
- Lógica de primer orden (aplicación a la especificación de propiedades que debe cumplir un programa).
- Realización de demostraciones por inducción.
- Diseñar programas pequeños mediante refinamientos sucesivos.
- Decidir qué ED elemental es más adecuado utilizar (vector, matriz, fichero...).

En EDI es muy importante observar la eficiencia de los algoritmos que manipulan estructuras de datos y comparar las distintas implementaciones de las estructuras de datos y elegir la que resulte más adecuada. De la misma manera, en MTP es importante el análisis de la eficiencia para comparar distintos algoritmos que resuelven el mismo problema. Por eso se incluyen varias implementaciones de las estructuras de datos y los costes, en cada una de las implementaciones, asociados a sus operaciones. Al igual que en las estructuras de datos, en los algoritmos también se incluye el código de los mismos y el coste comparativo con otros algoritmos.

En este tipo de asignaturas es muy importante el trabajo continuado propio y la realización de múltiples ejercicios y ejemplos para poner en práctica los conocimientos adquiridos y fomentar el encuentro de soluciones a los problemas tratados por estas asignaturas.

Por eso pensamos que esta herramienta será de gran utilidad y aportará grandes beneficios a los alumnos principiantes que se inicien en las asignaturas de EDI o MTP ya que su parte gráfica y visual ayuda a aclarar y comprender los conceptos.

No olvidemos que esta herramienta también puede servir de apoyo aquellos alumnos que estudian a distancia, por libre o con autoformación.

Tras cursar la asignatura de EDI se pretende que los alumnos adquieran unos conocimientos que esperamos que nuestra herramienta les ayude a adquirir:

- Analizar la eficiencia temporal y espacial de los algoritmos.
- Razonar sobre la corrección de los algoritmos.
- Diseñar y transformar algoritmos recursivos.
- Implementar tipos abstractos de datos con estructuras de datos y determinar cómo influyen en el coste de los programas.

Esta herramienta servirá de ayuda para el temario de EDI relacionado con las estructuras de datos, por lo que es recomendable que los alumnos hayan adquirido los conocimientos impartidos en EDI anteriormente (análisis de la eficiencia temporal y espacial, corrección de los algoritmos, diseño y transformación de algoritmos recursivos).

VEDYA ayudará a adquirir, y asentará los conocimientos que se adquieren tras cursar la asignatura de MTP:

- Análisis avanzado de la eficiencia de los programas.
- Transformación de algoritmos recursivos a iterativos.
- Resolver problemas con uno de los esquemas de resolución: divide y vencerás, método devorador, programación dinámica, ramificación y poda y precondicionamiento.
- Estudiar la complejidad de los programas y clasificarlos en base a ella.

3.- Descripción General de la Herramienta

3.1.- Ventana Principal

Al arrancar la aplicación aparece la interfaz de usuario donde se elige la estructura de datos o método algorítmico que se desee estudiar. Esta ventana no se cerrará hasta que cerremos la aplicación, para que el usuario pueda acceder a todas las estructuras de datos y métodos algorítmicos.

Figura 16 – Ventana principal de **VEDYA**

3.2.- Ventana Estructura de Datos

Hasta ahora las estructuras de datos disponibles son la Pila, la Cola y el Árbol Binario de Búsqueda. Al elegir cualquiera de estas, se abre la ventana correspondiente a la estructura de datos seleccionada. La aplicación se ha diseñado de tal forma, que todas las ventanas de estructuras de datos siguen un mismo patrón. Veamos en la **figura 17**, los elementos comunes a todas las ventanas de las estructuras de datos.

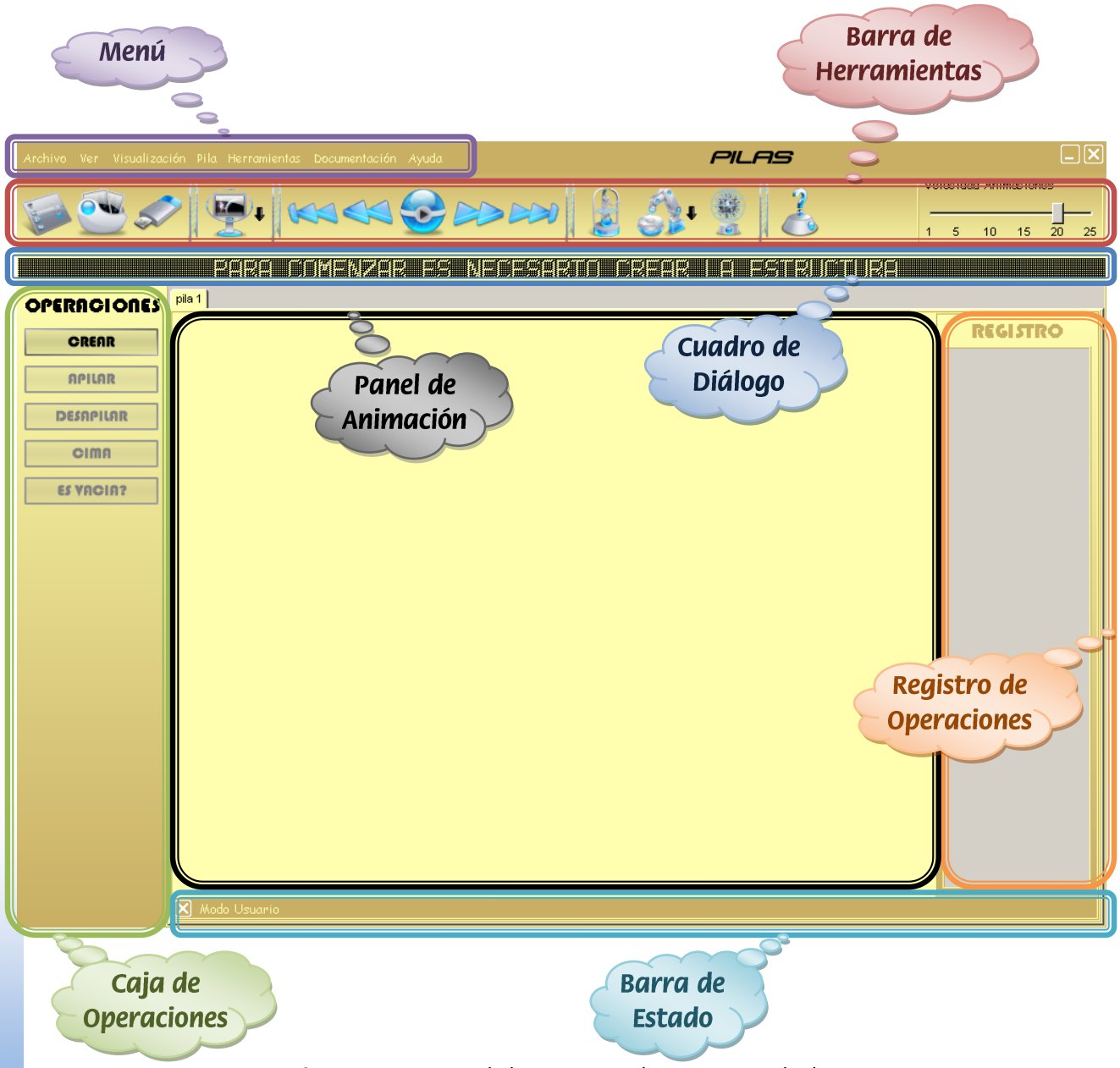


Figura 17 – Patrón de las ventanas de estructuras de datos

A continuación se muestra una breve descripción de cada uno de estos elementos comunes:

- **Menú**: encierra toda la funcionalidad de la herramienta.
- **Barra de herramientas**: se ha introducido en esta barra las principales acciones que se pueden realizar en la herramienta, para que el manejo de VEDYA sea fácil y ágil.
- **Cuadro de dialogo**: informa de la operación que se está realizando en el momento actual y muestra la información de las acciones que se realizan sobre la ventana.
- **Caja de operaciones**: contiene las operaciones que se pueden realizar sobre la estructura de datos. En cada momento, solo están activos los botones de las operaciones que se puedan ejecutar sobre la estructura sin llevar a esta a un estado de error, es decir, solo se permiten ejecutar operaciones validas para el estado actual de la estructura.
- Cada **pestaña** contiene una simulación de la estructura de datos. Dentro de cada pestaña nos encontramos:
  - **Panel de animación**: contiene la representación grafica de la estructura de datos y es donde se realizan las animaciones de las operaciones.
  - **Registro de operaciones**: contiene las operaciones que se han realizado hasta el momento sobre la estructura de datos. Representa el estado actual de la estructura de datos.
  - **Barra de estado**: muestra desde qué punto de vista se está visualizando la estructura de datos que contiene la pestaña. Además, contiene el botón para cerrar la pestaña.

3.3.- Funcionamiento Básico de VEDYA. Simulación de Estructuras de Datos

El objetivo principal de la VEDYA, es mostrar a los alumnos de EDI, el funcionamiento y las distintas formas de implementar las estructuras de datos más usuales, necesario para utilizarlas en la resolución de problemas de programación.

Para ello, se han representado y animado de manera gráfica, las estructuras de datos que ofrece la herramienta, de modo que el usuario pueda observar el resultado de aplicar las distintas operaciones sobre cada estructura.

Estas estructuras se muestran desde diferentes puntos de vista:

- **Modo usuario**: Desde este punto de vista se pretende mostrar al usuario las propiedades que caracterizan a la estructura de datos con una representación gráfica original, pero que refleje la filosofía de la estructura. Esta visualización es el primer contacto con la estructura, de hecho es la visualización seleccionada por defecto. En ella no queremos mostrar detalles de la implementación de la estructura de datos, sino que se entienda el comportamiento general de esta.
- **Modo implementación**: Desde este punto de vista se pretende mostrar al usuario las principales formas de implementar la estructura de datos. En estas visualizaciones se quiere enseñar de forma gráfica y teórica, mostrando el pseudocódigo de las operaciones en el cuadro de diálogo, los detalles de las distintas implementaciones, para su comprensión y aprendizaje. Se ha intentado que el usuario se dé cuenta de la eficiencia, en tiempo y espacio, de las distintas implementaciones.

3.3.1.- Pilas

Estructura de datos lineal, cuya característica principal es que el acceso a los elementos se realiza en orden inverso al de su almacenamiento. Se las suele denominar estructuras LIFO (Last In, First Out). La ventaja de las pilas es que el acceso a la estructura, tanto para su modificación como para la consulta de los datos almacenados, se realiza en un único punto (la cima de la pila), lo que facilita implementaciones sencillas y eficientes.

Las operaciones que se ofrecen al usuario son:

- Crear la pila vacía
- Apilar un elemento
- Desapilar el elemento de la cima
- Consultar el elemento de la cima
- Determinar si la pila está vacía

Sobre el panel de animación podrá verse de forma animada el comportamiento de aplicar estas operaciones. A continuación, se analiza cada una de ellas en detalle, independientemente de la visualización en la que se encuentre:

- **Crear**: al seleccionar esta operación se creará una pila vacía sobre la que se podrá aplicar el resto de las operaciones. Previamente, se deberá indicar el tipo de los elementos que contendrá la pila. Para ello, aparecerá una pequeña ventana, como la de la **figura 18**, donde se elegirá el tipo de los datos.

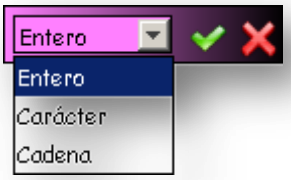


Figura 18 – Ventana donde el usuario elige el tipo de la estructura

Una vez que se ha ejecutado la operación, se deshabilitará su botón, ya que no se podrá crear la pila por segunda vez.

- **Apilar**: para que esta operación pueda estar disponible será necesario que la pila se haya creado con anterioridad. Al seleccionar esta operación podrá verse de manera animada cómo se introduce un nuevo elemento en la pila.

  Es el usuario el que indica el dato que se introduce, siendo éste del tipo que se indicó al crear la pila. Para ello, aparecerá una ventana, como la que se muestra en la **figura 19**, donde se podrá introducir el dato. Solo se permite introducir los caracteres que formen datos del tipo indicado en la operación crear, por ejemplo, si al crear se indico que los datos fuesen enteros solo se permiten introducir los caracteres del 0 al 9 y "–". Además, por razones de espacio del elemento, solo se permiten, como máximo, elementos de longitud 3.

Figura 19 – Ventana donde el usuario introduce los datos de entrada de la estructura de datos

  Por limitaciones del espacio de la pantalla, hemos considerado que, en cada una de las vistas de las pilas haya un máximo de 52 elementos, siendo este un tamaño suficientemente razonable, como para poder entender el comportamiento de las pilas. Entonces, cuando se inserte el elemento número 52, el botón de la operación apilar se deshabilitará.

- **Desapilar**: al seleccionar esta operación podrá verse de manera animada cómo se elimina el elemento de la cima de la pila. Para que el botón de esta operación esté habilitado es necesario que se haya creado la pila y que contenga algún elemento.
- **Cima**: al seleccionar esta operación se devolverá el valor del elemento de la cima de la pila, y se señalará de forma visual el elemento. Para que ésta operación esté disponible será necesario que la pila se haya creado y que contenga algún elemento.
- **Es vacía?**: si se selecciona esta operación, la aplicación indicará si la pila está vacía o si contiene algún elemento. La pila estará vacía nada mas crearla o cuando se hayan desapilado todos sus elementos. Para que el botón de esta operación esté habilitado es necesario que se haya creado la pila.

A continuación se muestra como se representa gráficamente las pilas para cada una de las visualizaciones y las animaciones que se realizan.

3.3.1.1.- Visualización "Modo Usuario"

En esta visualización queremos mostrar la filosofía de las pilas, es decir, que los elementos se van a ir insertando a continuación del último que está en la pila, y no en otro orden, y que solo es accesible el elemento que se encuentra en la cima de esta. Para ello:

- **Crear**: al seleccionar esta operación aparecerá la pila vacía, representada mediante una tubería, como la que se muestra en la **figura 20**. Se puede observar que está tubería está diseñada con una sola abertura, para que la inserción y eliminación de los elementos se produzca por un único punto.

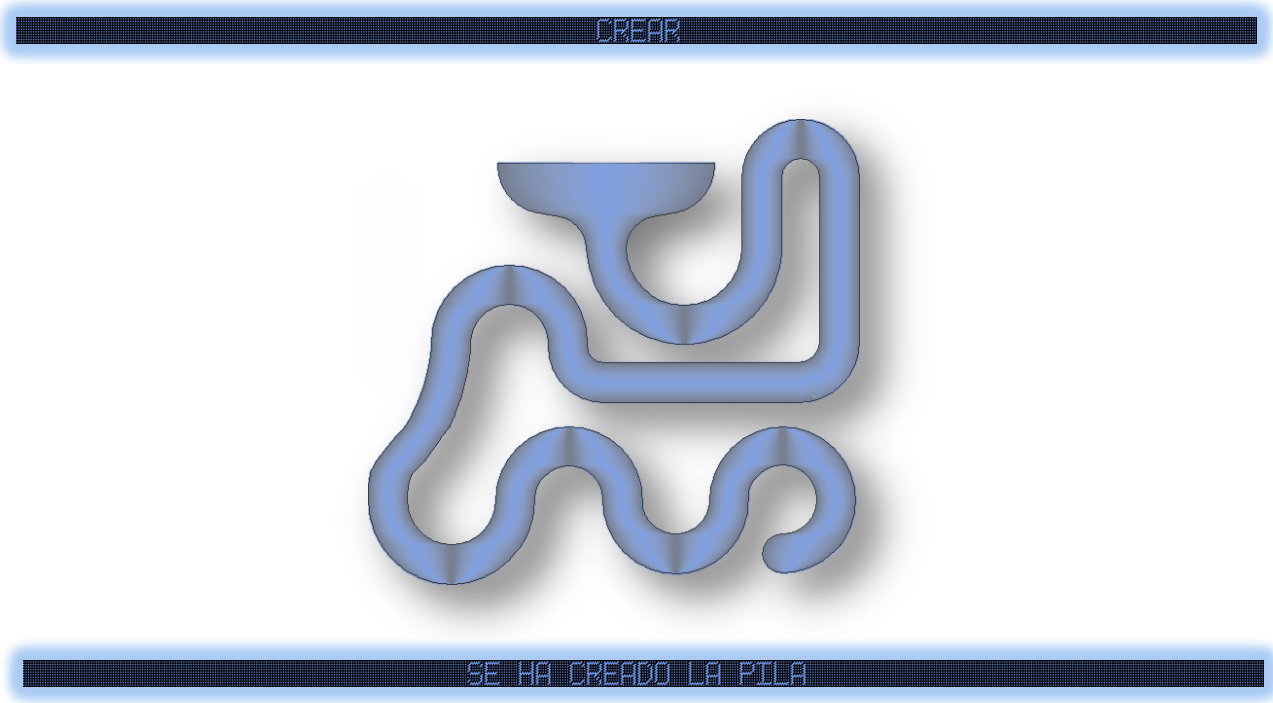
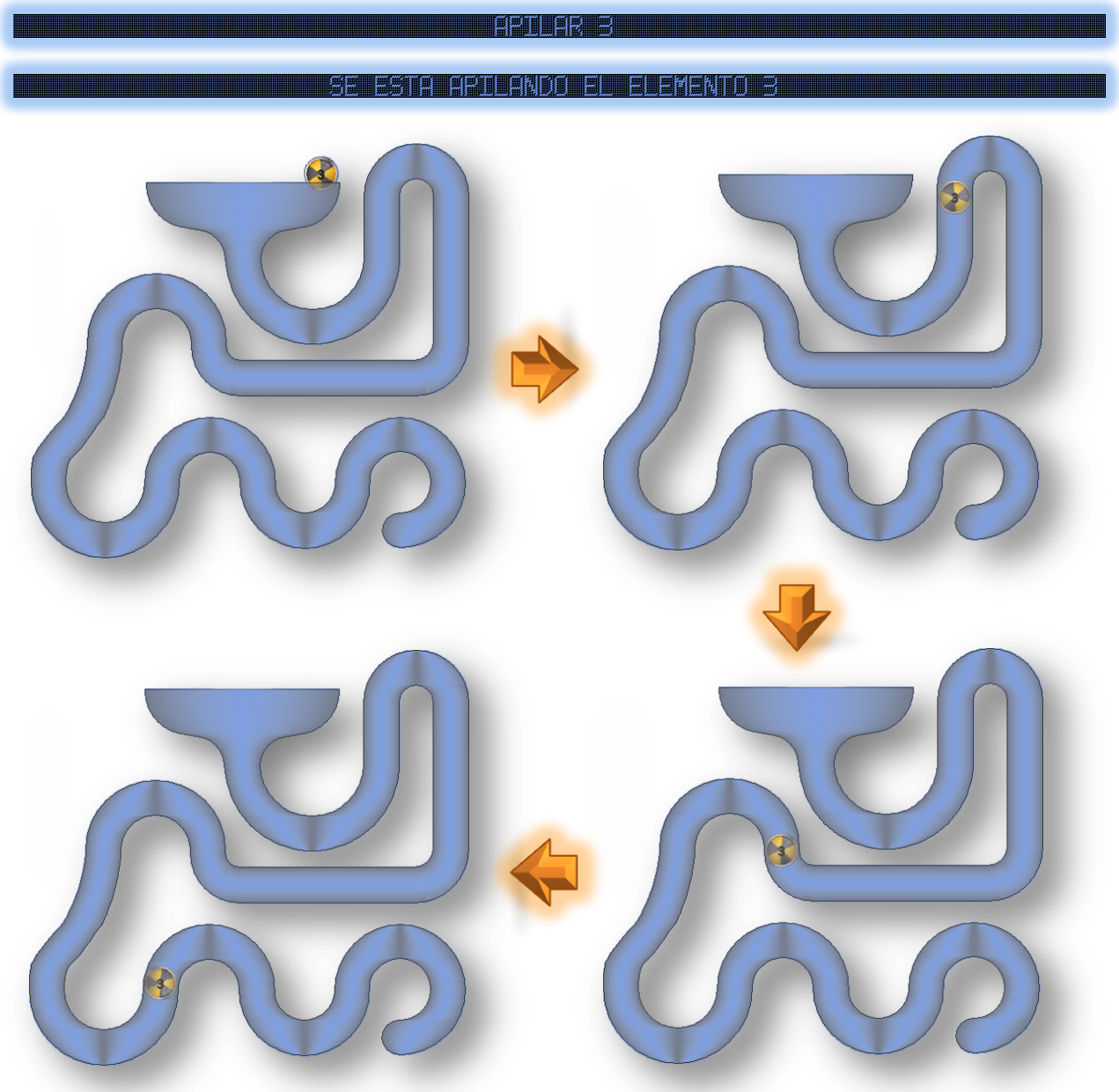


Figura 20 – Animación de la operación Crear. Pila. Visualización de Usuario.

- **Apilar:** *al seleccionar esta operación, y después de haber indicado el dato que queremos introducir, podrá verse como aparece el nuevo elemento en la parte superior del panel de animación y cae rodando por toda la tubería hasta que se choque con el final de esta o con el último elemento. Este elemento, al ser el último es el único al que podremos acceder, ocupará la cima de la pila. En la* ***figura 21*** *se muestra la secuencia que se produce al pulsar la operación apilar.*



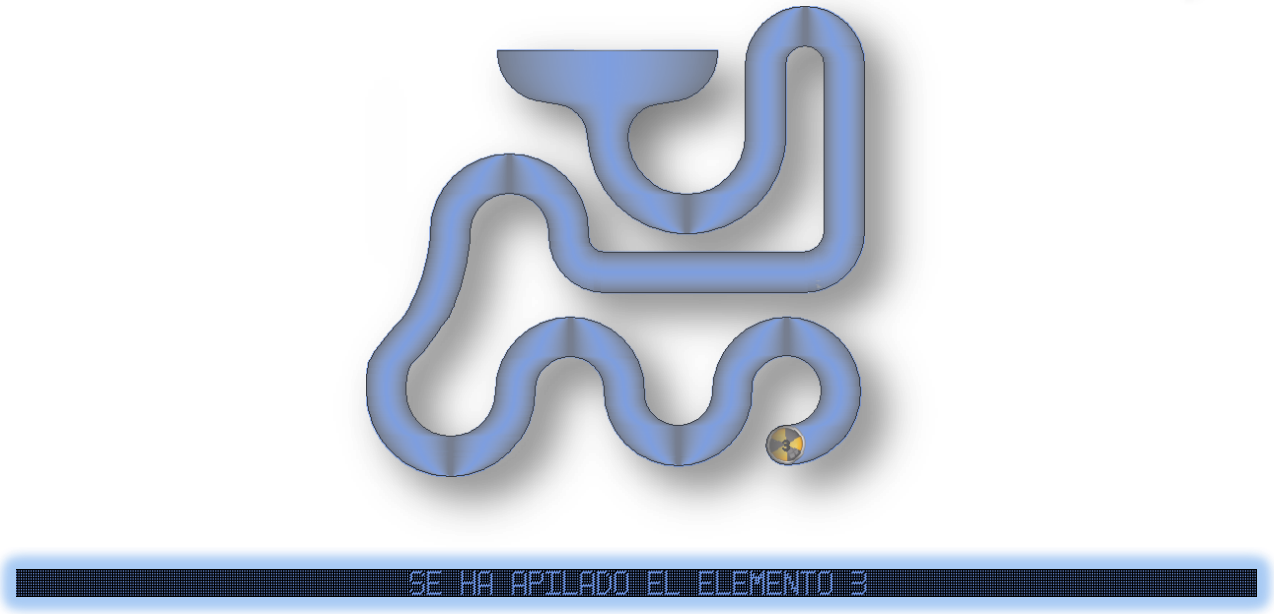


Figura 21 – Animaciones de la operación apilar el elemento 3. Pila. Visualización de Usuario.

Para continuar la explicación del resto de operaciones de la pila, Se han apilado los siguientes elementos en orden:

65, 4, 8, 52, 35, 98, 73, 37, 69, 47, 1, 84,
79, 63, 25, 18, 43, 29, 30, 84, 91, 56, 37

Después de apilarlos, la pila queda como se muestra en la **figura 22**.

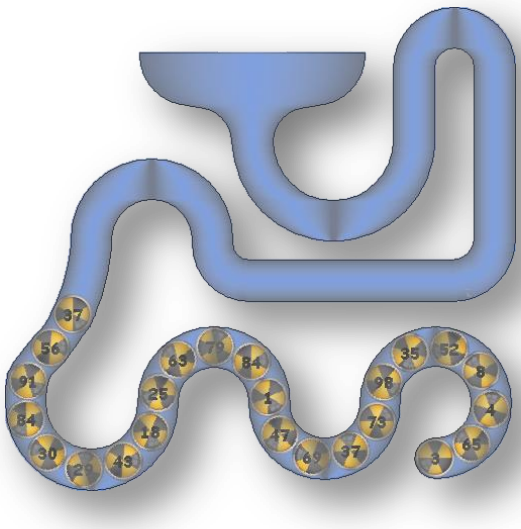
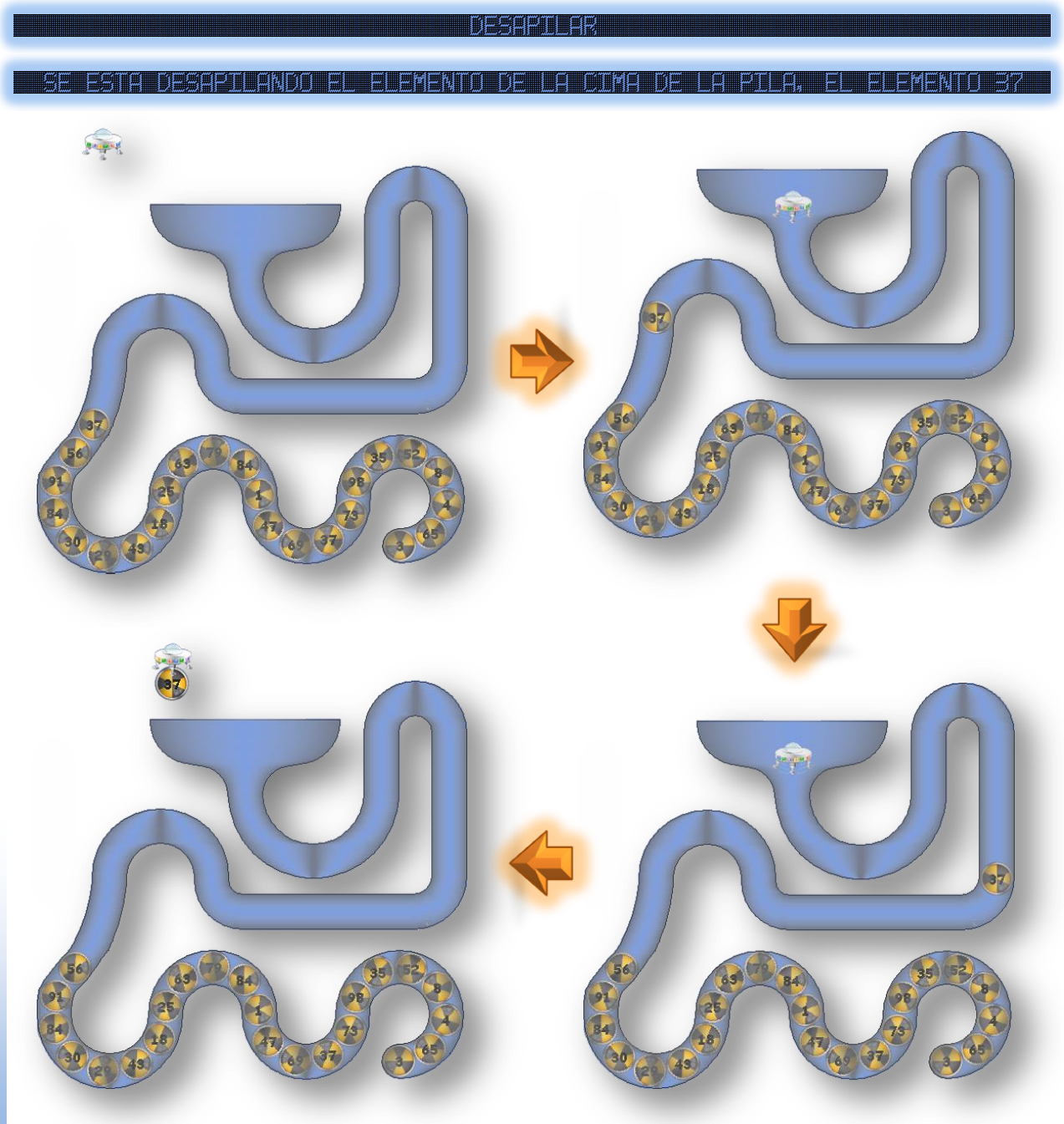


Figura 22 – Apilamos varios elementos. Pila. Visualización de Usuario.

- **Desapilar**: al seleccionar esta operación aparece un OVNI por la parte superior del panel de animación con intención de llevarse un elemento. Para ello se coloca en la boca de la tubería y abduce el único elemento posible, el que se encuentra en la cima de la pila. Una vez abducido se lo lleva fuera de la pantalla (ver **figura 23**).



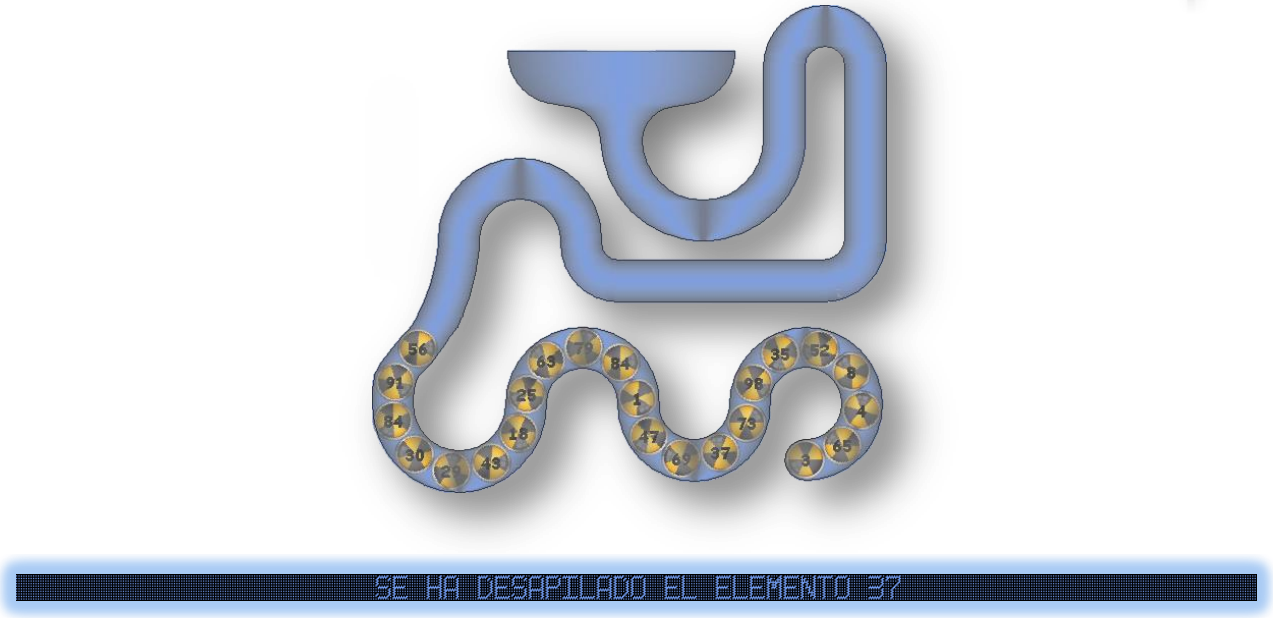


Figura 23 – Animaciones de la operación desapilar el elemento 37. Pila. Visualización de Usuario.

- ***Cima***: al seleccionar esta operación aparecen 4 flechas que señalan el elemento que se encuentra en la cima de la pila (ver **figura 24**).

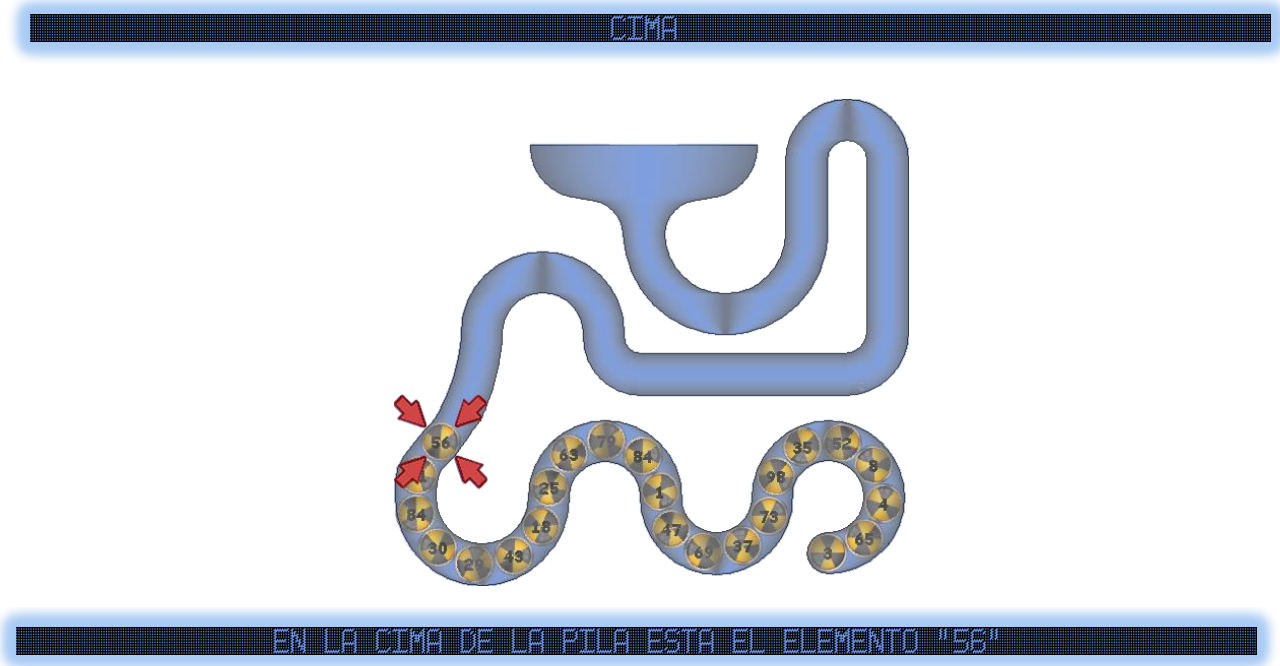


Figura 24 – Animación de la operación Cima. Pila. Visualización de Usuario.

- **Es Vacía?**: *al seleccionar esta operación se muestra en el cuadro de dialogo si la pila está o no vacía (ver* ***figuras* 25** y **26**).

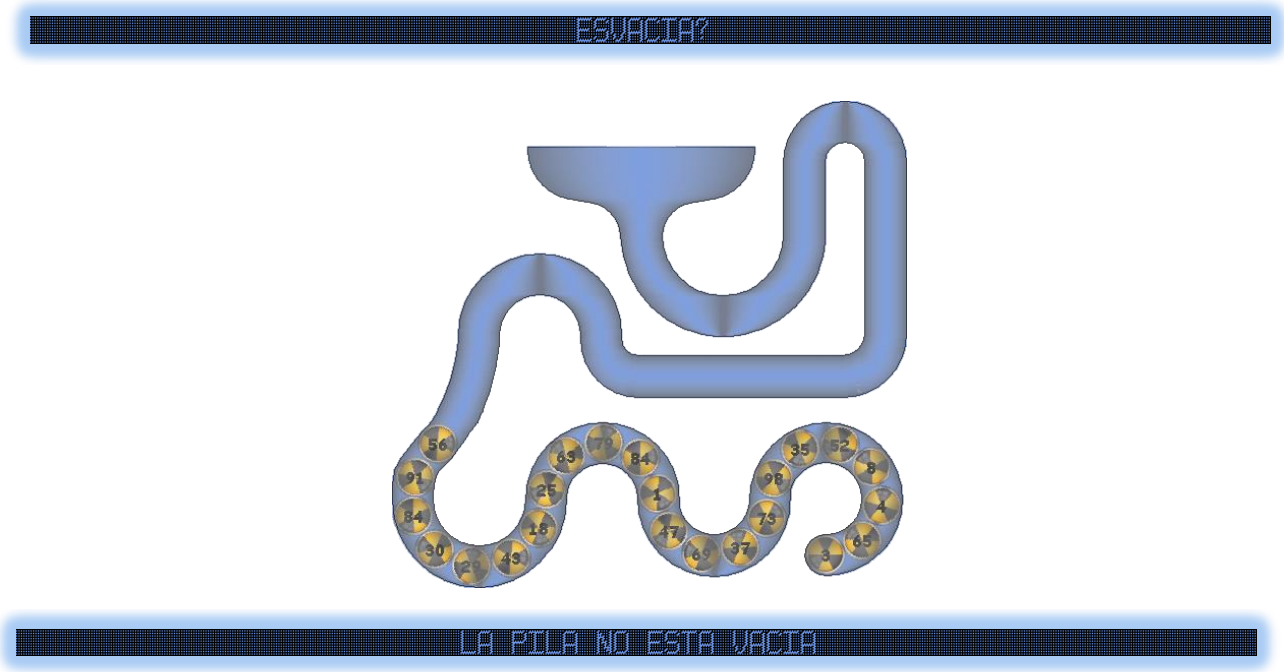


Figura 25 – Animación de la operación EsVacia?. Pila no vacía. Visualización de Usuario.

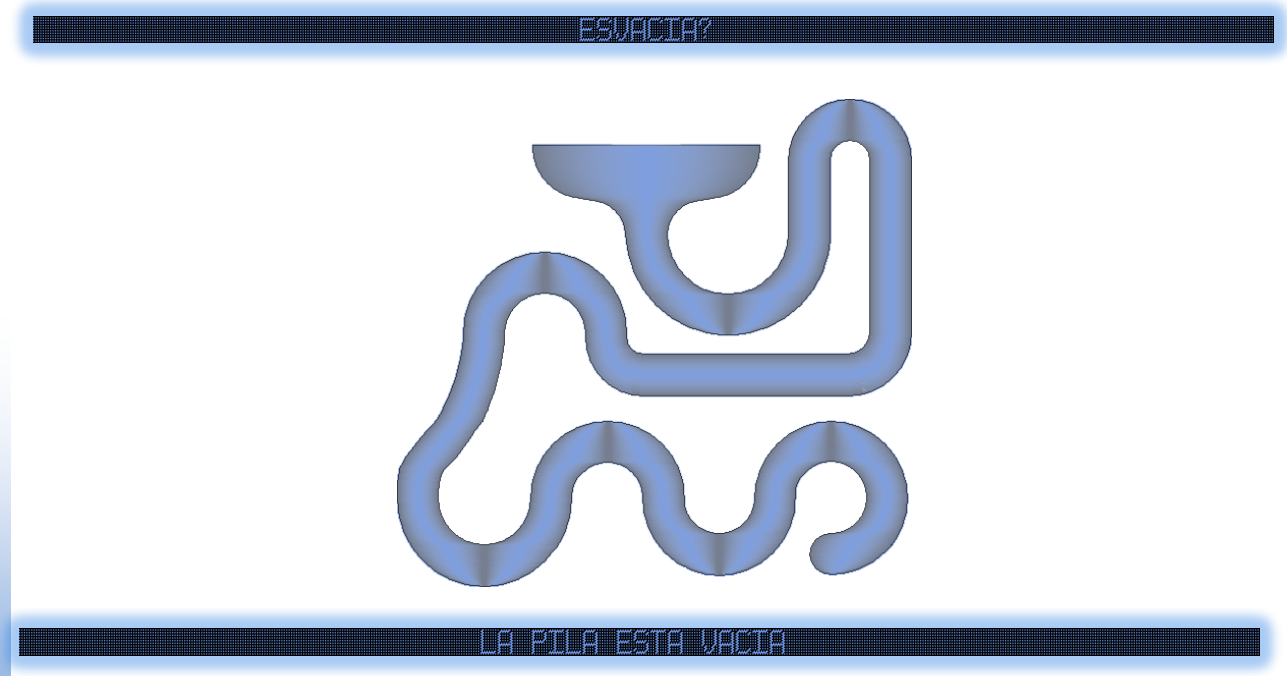


Figura 26 – Animación de la operación EsVacia?. Pila vacía. Visualización de Usuario.

3.3.1.2.- Visualización "Modo Implementación Estática"

En esta visualización se mostrará la representación estática de las pilas utilizando un vector. La idea es utilizar un vector donde vamos colocando los elementos de la pila, en orden creciente de posiciones según se van apilando, y un índice que apuntará a la cima. Esta representación tiene el inconveniente de que el tamaño de la estructura queda restringido por la capacidad del soporte de almacenamiento, en este caso por el tamaño del vector.

- ***Crear***: al seleccionar está operación aparecerá la estructura que representa el vector de 52 elementos, y el índice-cima inicializado a 0, ya que la pila se crea vacía. Al realizar la operación crear sobre una pila representada estáticamente, se tiene que crear un vector, de forma que se reserva un espacio en memoria correspondiente al tamaño del vector. En la **figura 27**, se muestra el vector que representa la pila, dividido en 4 trozos de 13 elementos cada uno (el tamaño limitado de la pantalla nos ha hecho representar el vector de esta forma).

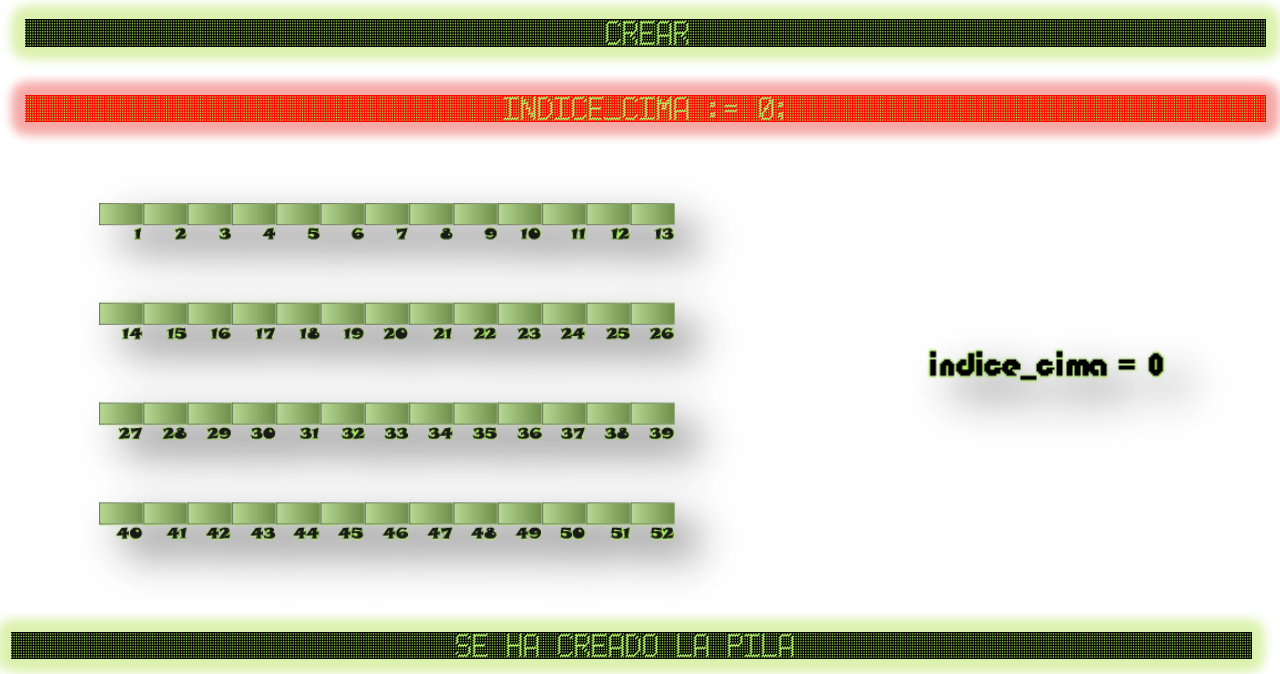
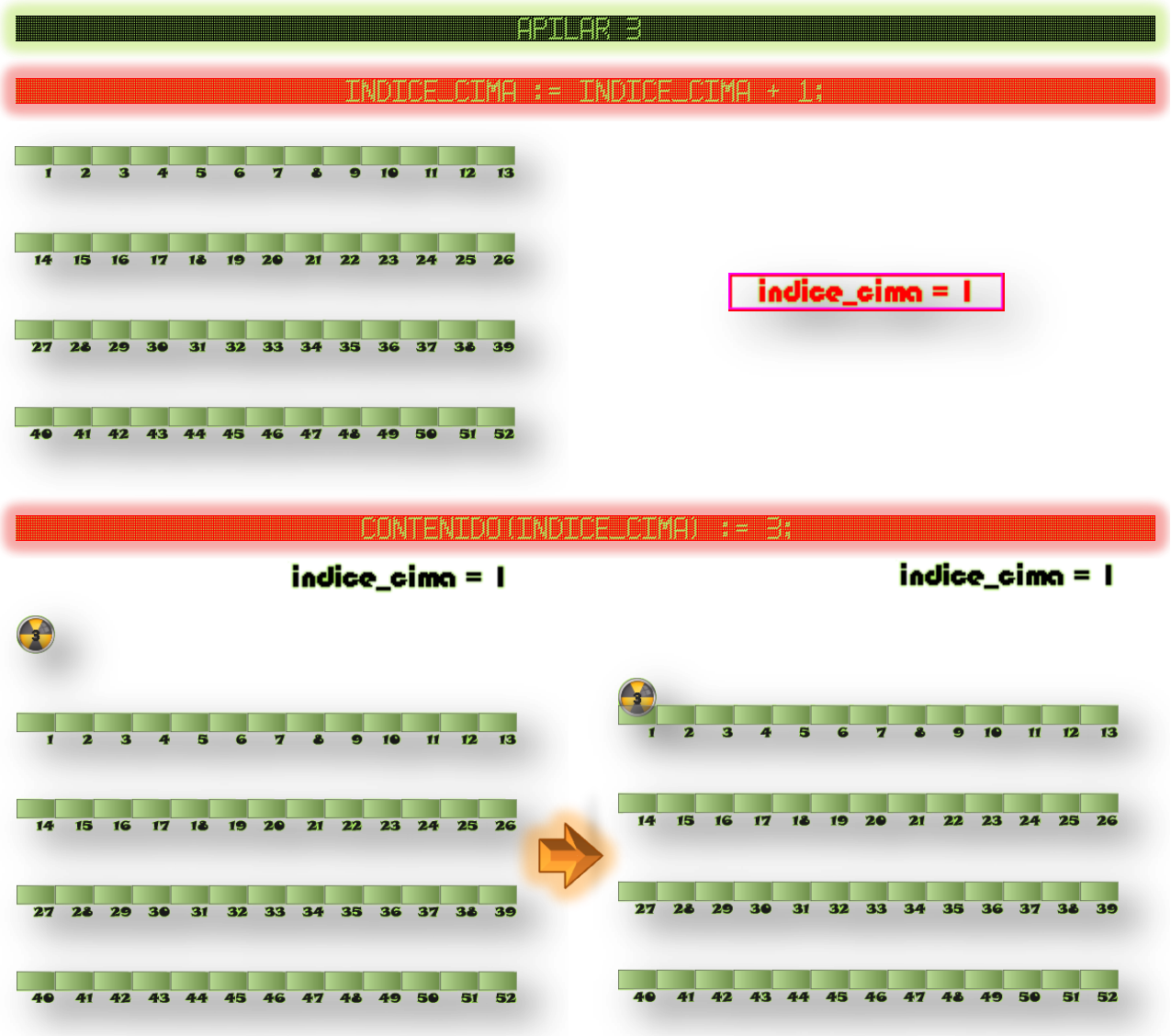


Figura 27 – Animación de la operación Crear. Pila. Visualización Estática.

- **Apilar**: *al seleccionar esta operación, y después de haber indicado el dato que queremos introducir, se mostrará de forma animada cómo se introduce un nuevo elemento en la pila. Se aumentará el índice-cima en una unidad y se introducirá el elemento en dicha posición del vector. (Ver* ***figura 28***)



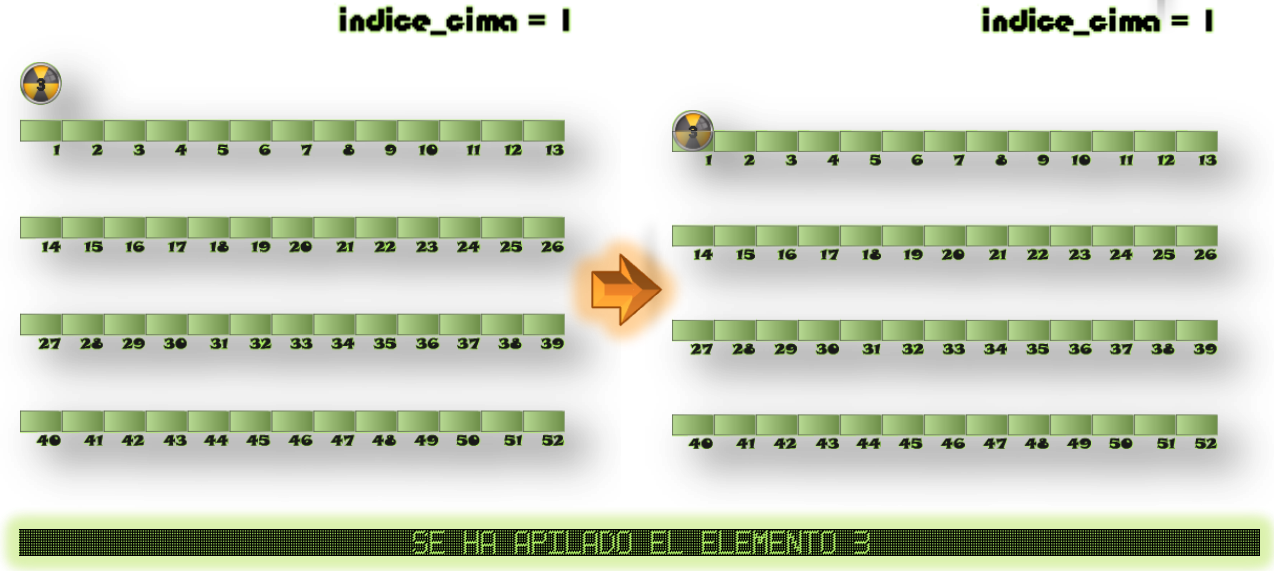


Figura 28 – Animaciones de la operación apilar el elemento 3. Pila. Visualización Estática.

Para continuar la explicación del resto de operaciones de la pila, se han apilado los siguientes elementos en orden:

65, 4, 8, 52, 35, 98, 73, 37, 69, 47, 1, 84,
79, 63, 25, 18, 43, 29, 30, 84, 91, 56, 37

Después de apilarlos, la pila queda como se muestra en la **figura 29**.

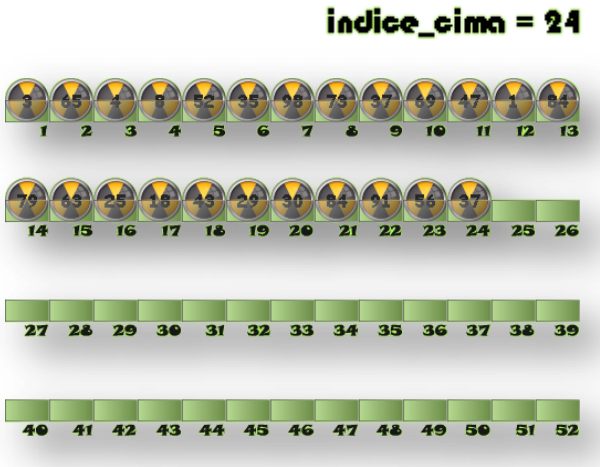


Figura 29 – Apilamos varios elementos. Pila. Visualización Estática.

- **Desapilar**: al seleccionar esta operación, se disminuye índice-cima en una unidad, apuntando a la nueva cima. Para una mayor claridad eliminamos dicho elemento del vector, aunque realmente se quede como basura dentro del vector, (ver secuencia de la **figura 30**).

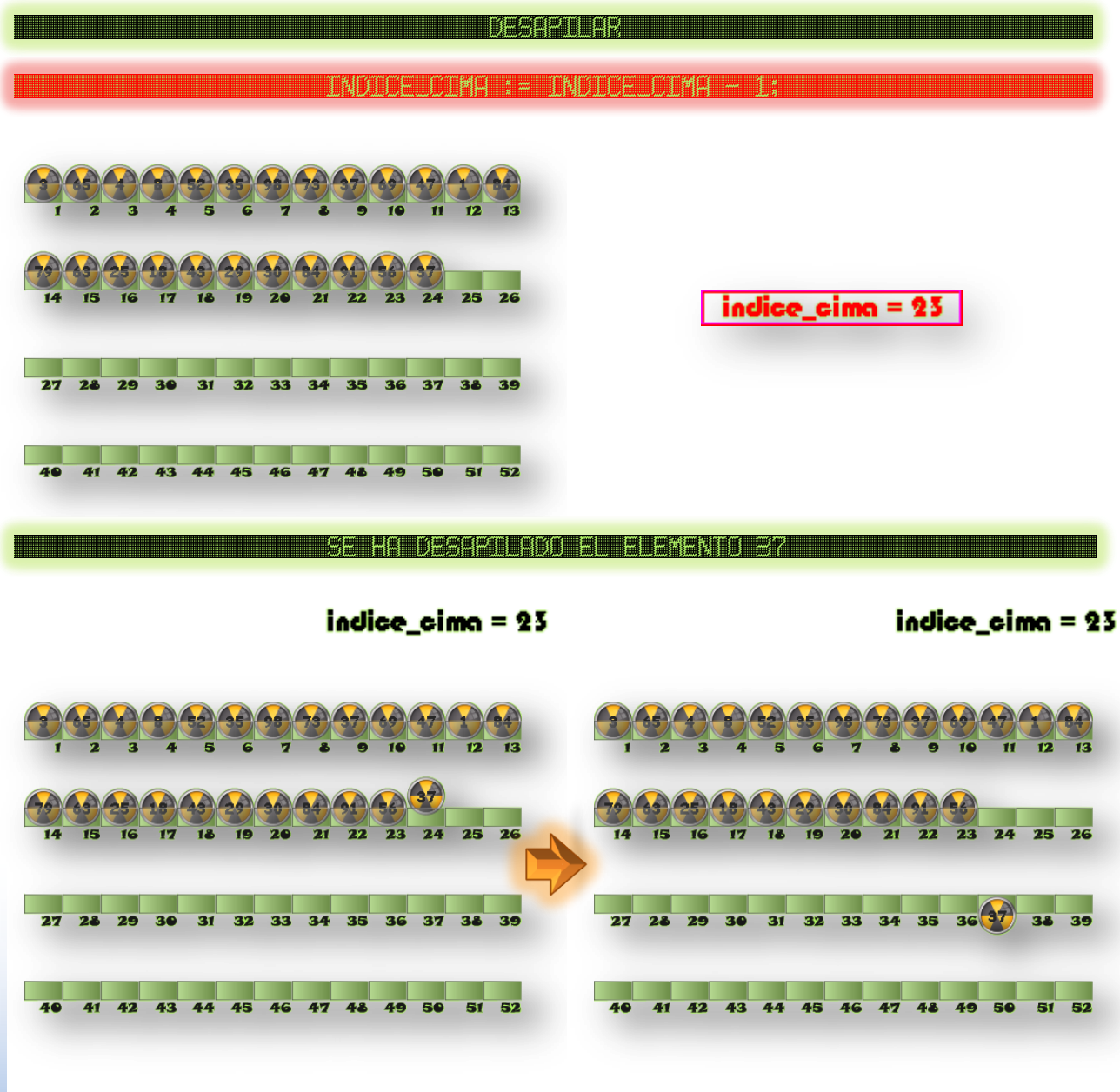


Figura 30 – *Animaciones de la operación desapilar el elemento 37. Pila. Visualización Estática.*

- ***Cima***: *al seleccionar esta operación aparecen 4 flechas que señalan el elemento que está contenido en la posición índice-cima del vector, (ver* ***figura 31****).*

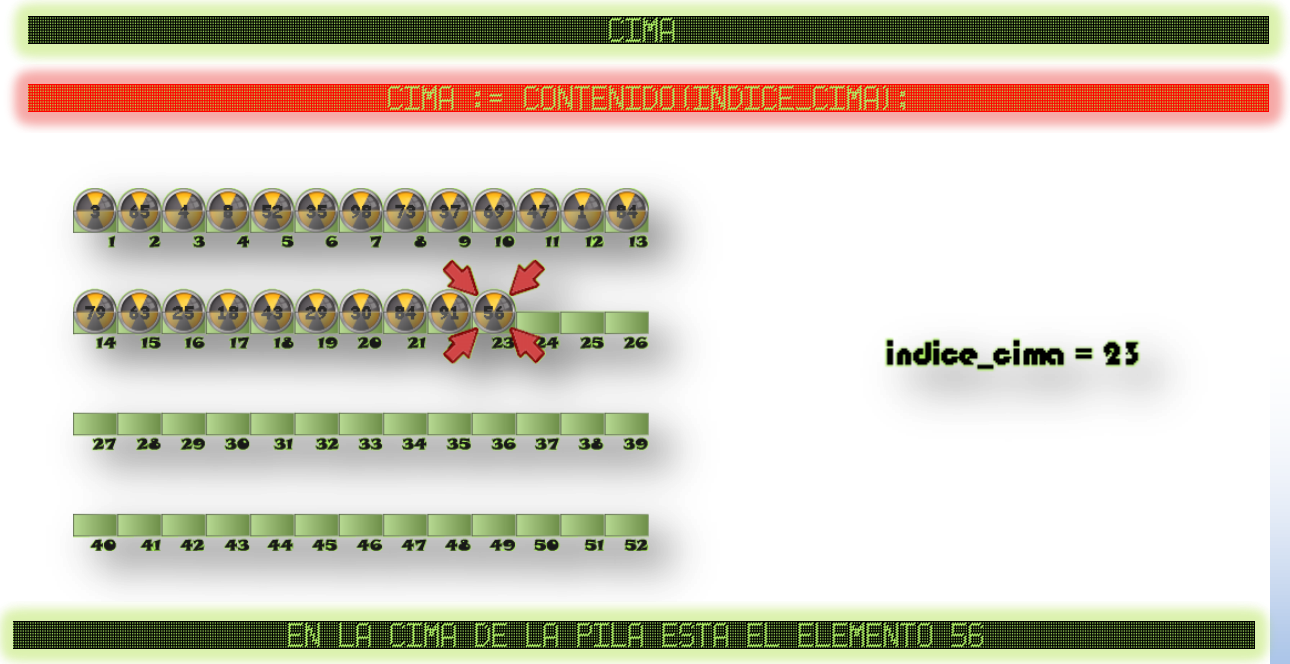


Figura 31 – *Animación de la operación Cima. Pila. Visualización Estática.*

- **Es vacía?**: *al seleccionar esta operación se muestra en el cuadro de dialogo si la pila está o no vacía, es decir, si índice-cima es distinto o igual a cero, respectivamente, (ver* ***figuras*** **32** y **33**).

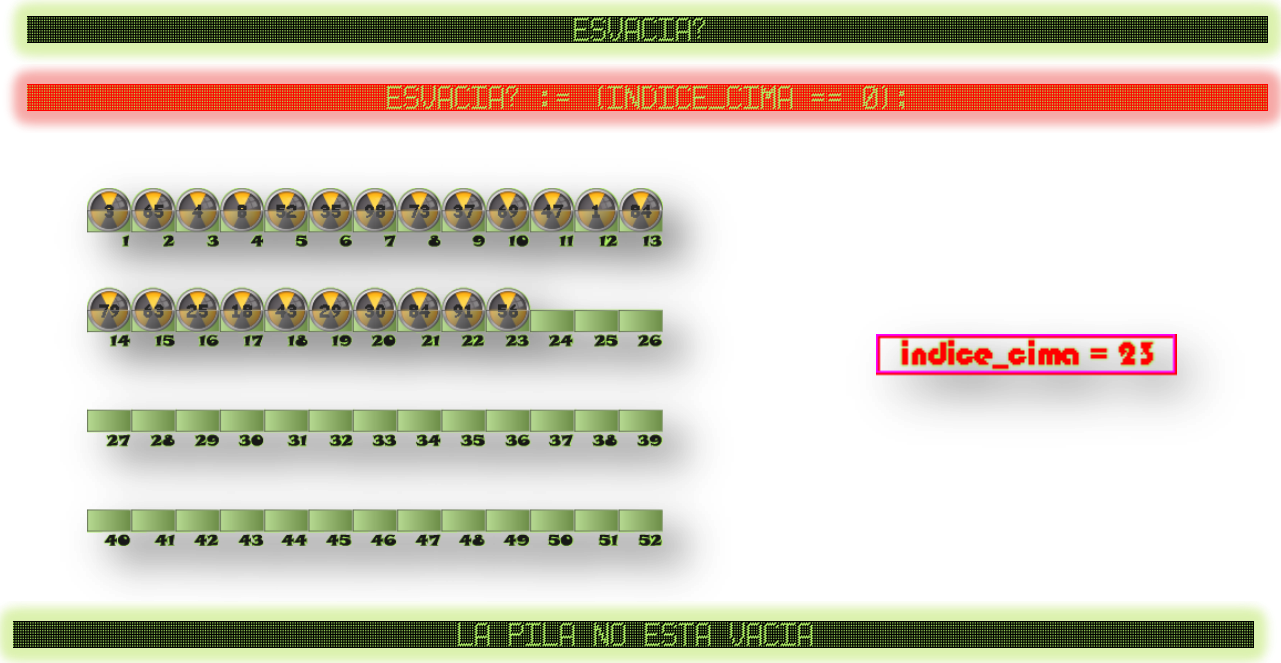


Figura 32 – *Animación de la operación EsVacia?. Pila no vacía. Visualización Estática.*

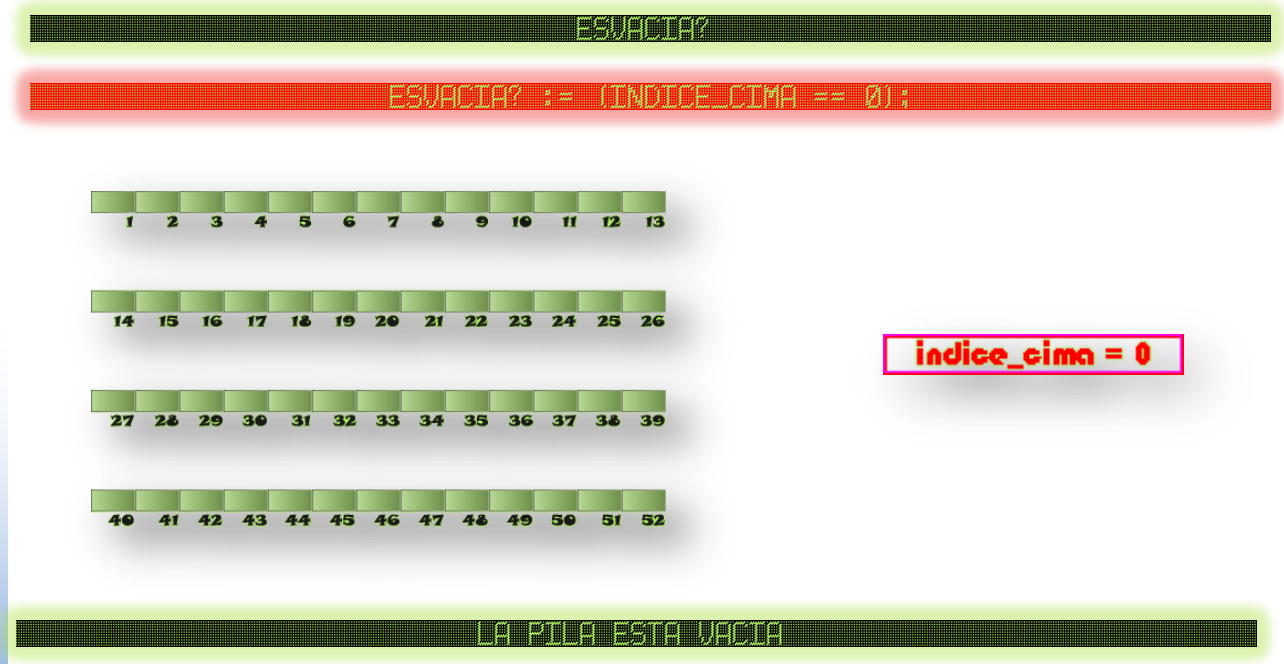


Figura 33 – *Animación de la operación EsVacia?. Pila vacía. Visualización Estática.*

3.3.1.3.- VISUALIZACIÓN "MODO IMPLEMENTACIÓN DINÁMICA"

En esta visualización se mostrará la representación dinámica de las pilas utilizando punteros. Esta representación se realiza mediante una estructura lineal enlazada, de forma que la cima corresponda al extremo directamente accesible de la estructura mediante el puntero **P**, con lo que se consigue que el coste en tiempo de todas las operaciones sea constante.

- ***Crear***: al seleccionar esta operación el puntero **P** no apunta a ninguna estructura, apunta a **"null"**, representado como una toma de tierra, (ver ***figura* 34**).

***Figura* 34** – Animación de la operación Crear. Pila. Visualización Dinámica.

- ***Apilar***: al seleccionar esta operación, y después de haber indicado el dato que queremos introducir, se mostrará de forma animada cómo se introduce un nuevo elemento en la pila. Para ello es necesario utilizar el puntero auxiliar **Q**. La secuencia de animaciones será la siguiente:
  - Se reserva espacio en memoria para el elemento apuntado por el puntero **Q**.
  - El nuevo elemento apuntará a lo que apunta **P**.
  - El puntero **P** apuntará a lo que apunta **Q**.

  (Ver las ***figuras* 35** y **36**)

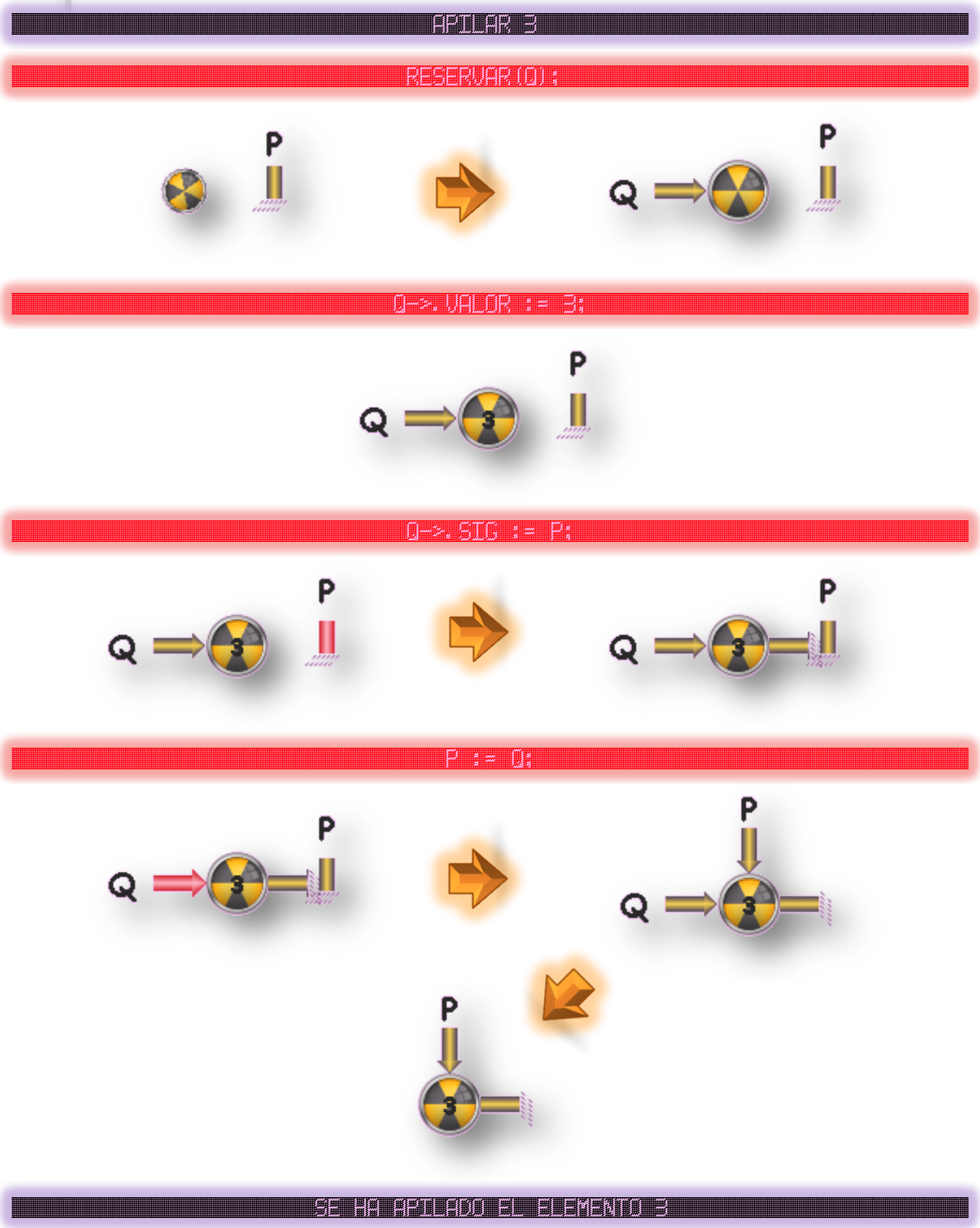


Figura 35 – Animación de la operación Apilar el elemento 3. Pila. Visualización Dinámica.

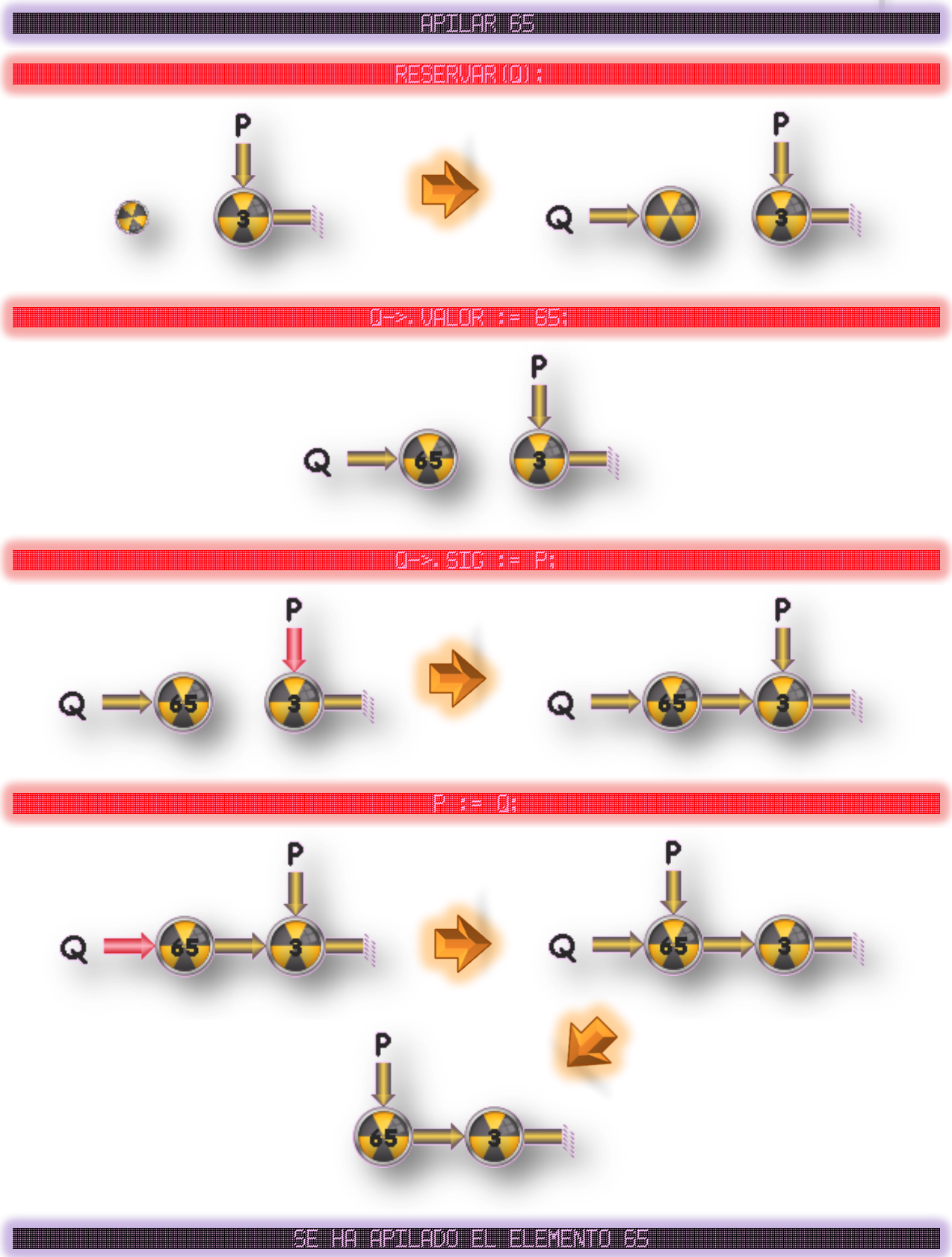


Figura 366 – Animación de la operación Apilar el elemento 65. Pila. Visualización Dinámica.

Para continuar la explicación del resto de operaciones de la pila, se han apilado los siguientes elementos en orden:

4, 8, 52, 35, 98, 73, 37, 69, 47, 1, 84, 79,
63, 25, 18, 43, 29, 30, 84, 91, 56, 37

Después de apilarlos, la pila queda como se muestra en la **figura 37**.

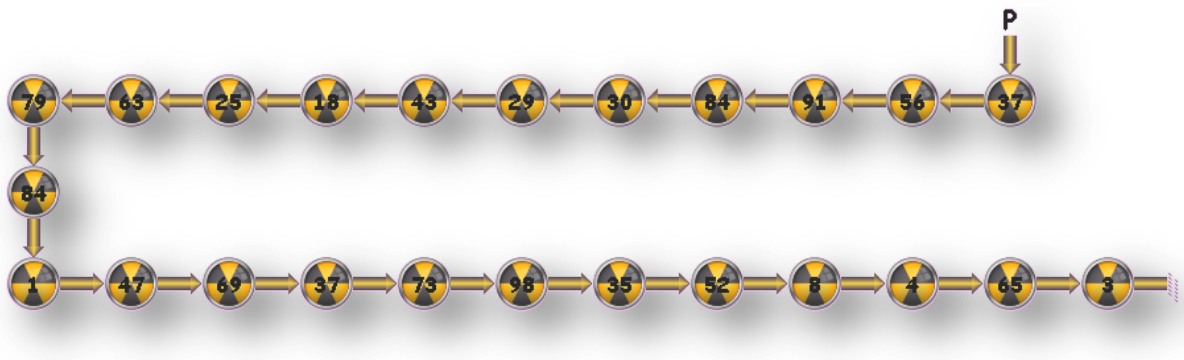


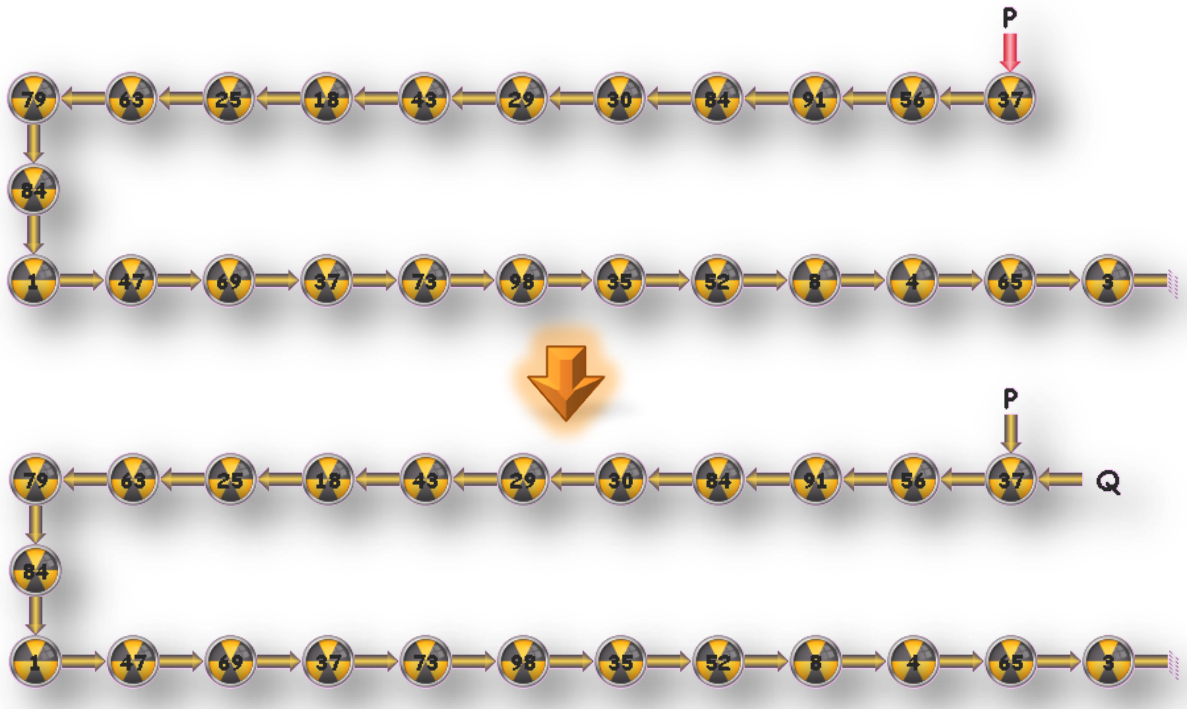
Figura 37 – Apilamos varios elementos. Pila. Visualización Dinámica.

- **Desapilar**: al seleccionar esta operación podrá verse de manera animada cómo se elimina el nodo apuntado por **P**. Para ello es necesario utilizar el puntero auxiliar **Q**. La secuencia de animaciones será la siguiente:
  - **Q** apuntará a lo que apunta **P**.
  - **P** apuntará al siguiente de **Q**.
  - Se libera **Q**.

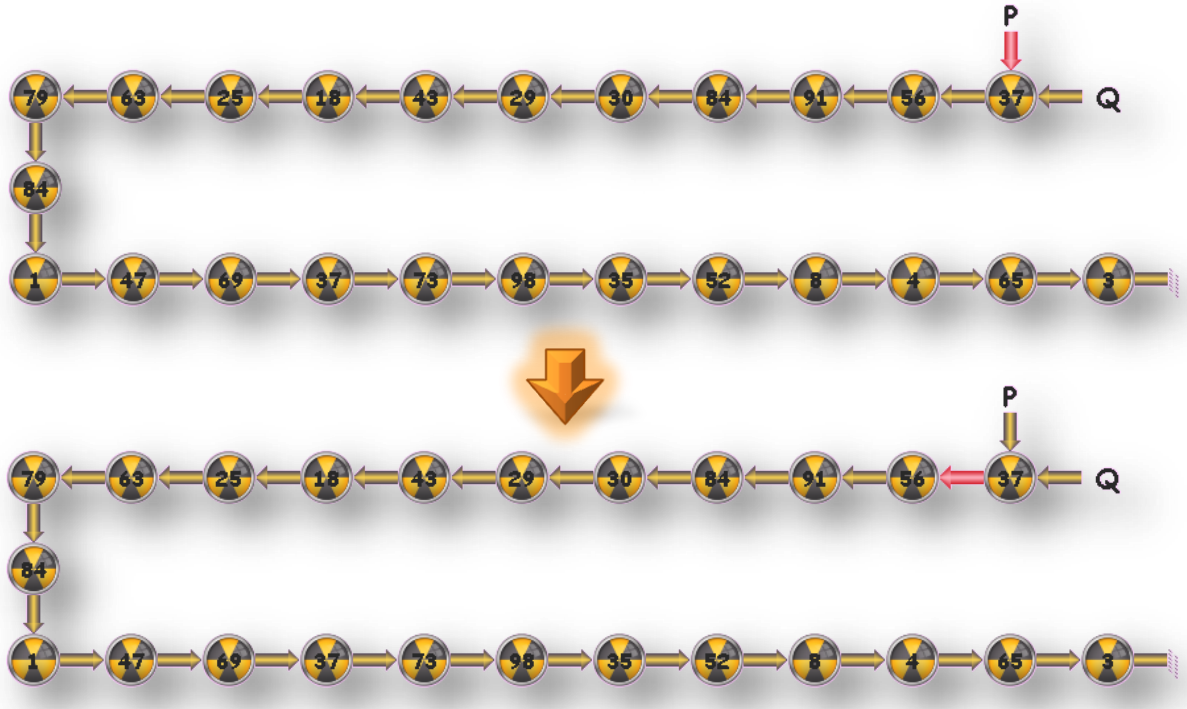
  (Ver **figura 38**)

DESAPILAR

```
Q := P;
```




```
P := P->.SIG;
```



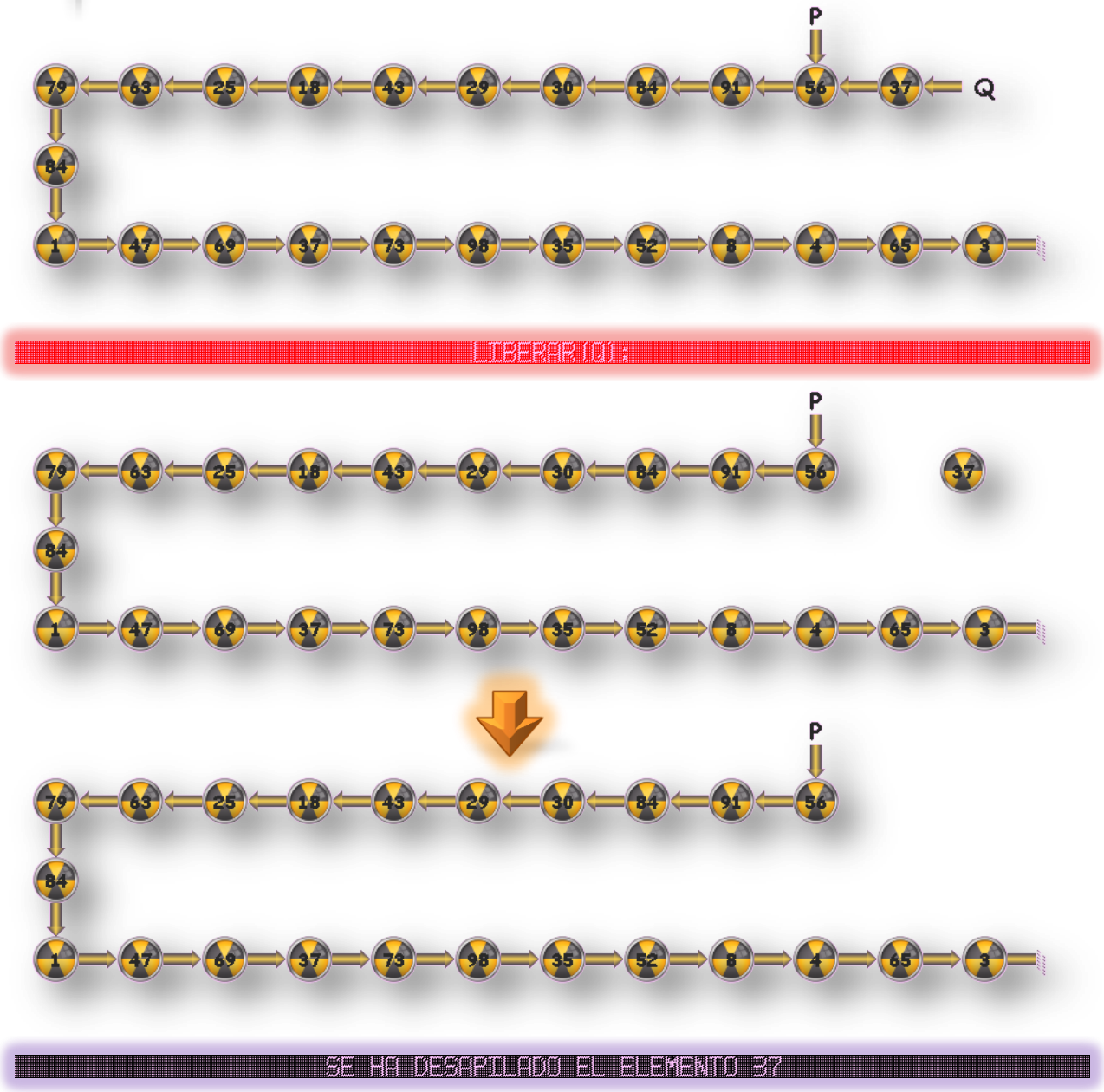


Figura 38 – Animación de la operación Desapilar el elemento 37. Pila. Visualización Dinámica.

- ***Cima***: *al seleccionar esta operación aparecen 4 flechas que señalan el elemento al que apunta* **P**, *(ver* ***figura* 39**).

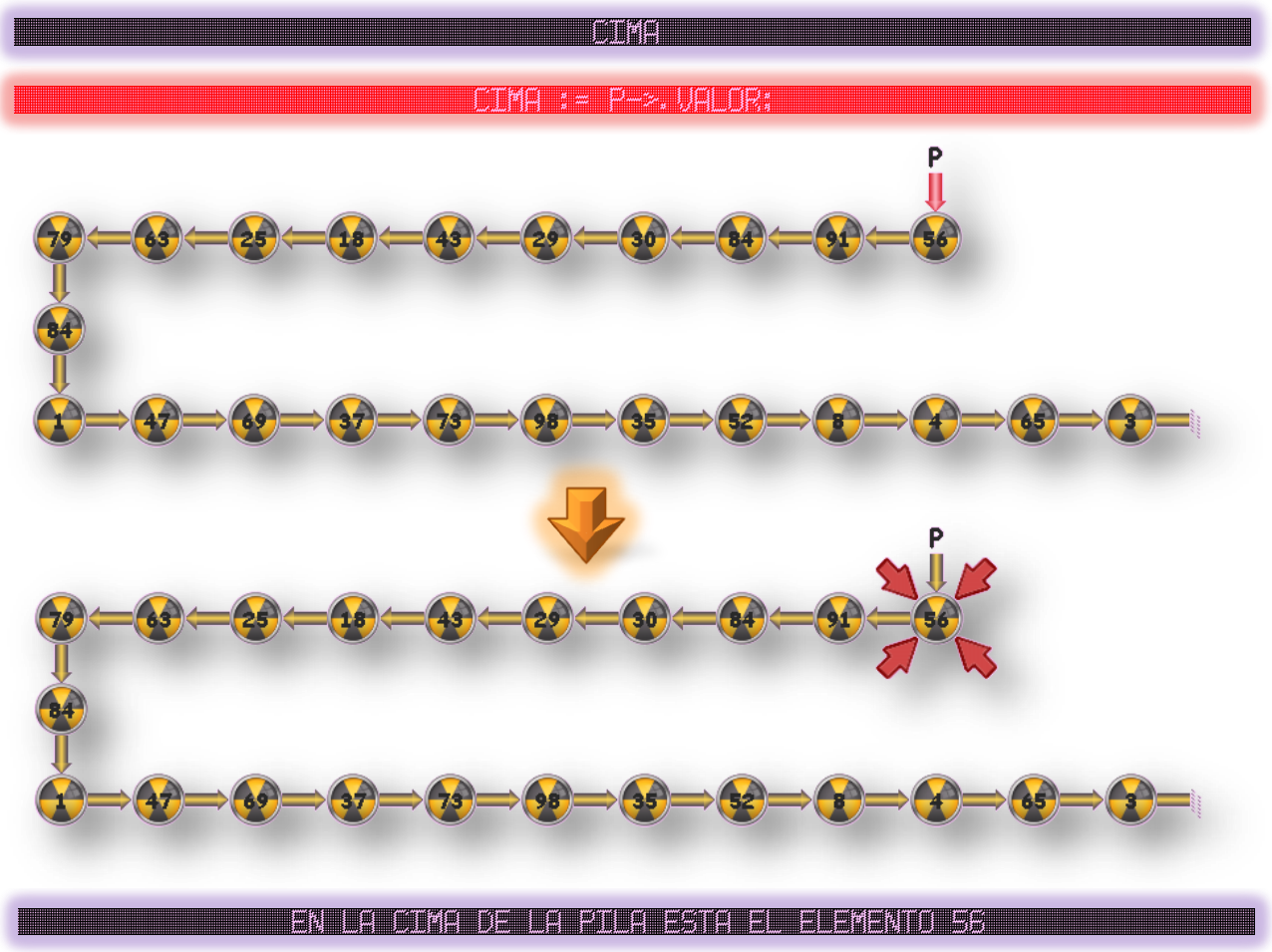


***Figura* 39** – *Animación de la operación Cima. Pila. Visualización Dinámica.*

- **Es vacía?**: *al seleccionar esta operación se muestra en el cuadro de dialogo si la pila está o no vacía, es decir, si* **P** *apunta a* **"null"** *o no, respectivamente, (ver* ***figuras*** **40** y **41**).

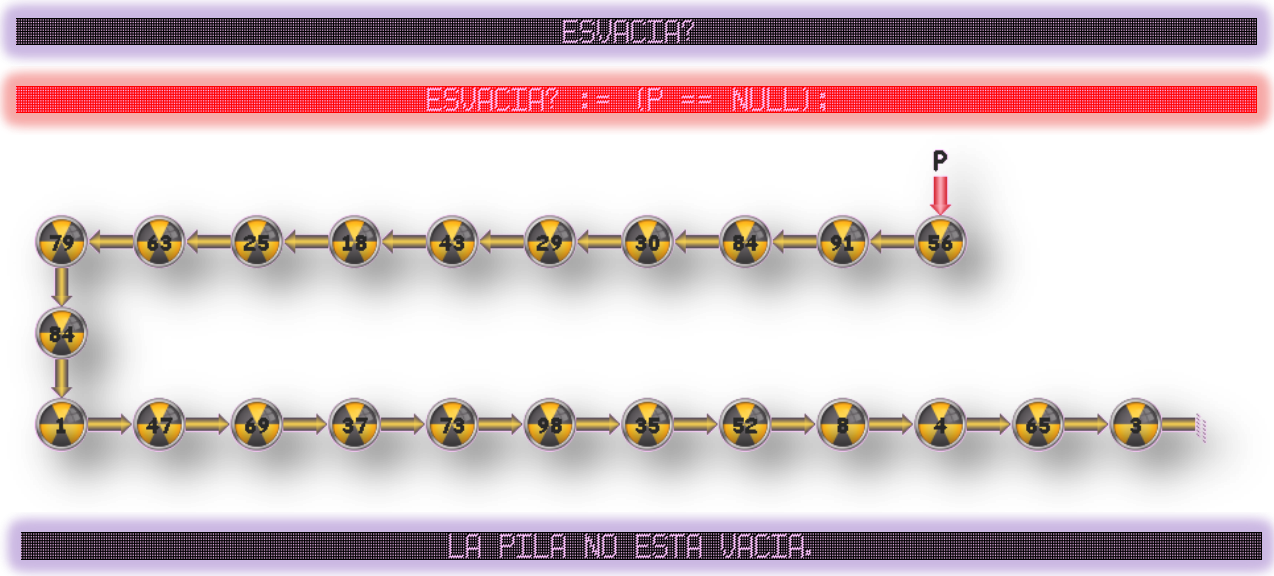


Figura 40 – *Animación de la operación EsVacia?. Pila no vacía. Visualización Dinámica.*

Figura 41 – *Animación de la operación EsVacia?. Pila vacía. Visualización Dinámica.*

3.3.2.- Colas

Estructura de datos lineal, cuya característica principal es que el acceso a los elementos se realiza en el mismo orden en que fueron almacenados. Se las suele denominar estructuras FIFO (First In, First Out). A diferencia de las pilas (vista en la **sección 3.3.1**) que tienen un único punto de acceso, las colas presentan dos zonas de interés:

- El extremo final, por donde se incorporan los elementos
- La cabecera, por donde se consultan y se eliminan los elementos

Las operaciones que se ofrecen al usuario son:

- Crear la cola vacía
- Añadir un elemento al final de la cola (Pedir vez)
- Eliminar el primer elemento de la cola (Avanzar)
- Consultar el primer elemento
- Determinar si la cola está vacía

Sobre el panel de animación podrá verse de forma animada el comportamiento de aplicar estas operaciones. A continuación, se analiza cada una de ellas en detalle, independientemente de la visualización en la que se encuentre:

- ***Crear***: al seleccionar esta operación se creará una cola vacía sobre la que se podrá aplicar el resto de las operaciones. Previamente, se deberá indicar el tipo de los elementos que contendrá la cola. Para ello, aparecerá una pequeña ventana, como la de la ***figura* 42**, donde se elegirá el tipo de los datos.

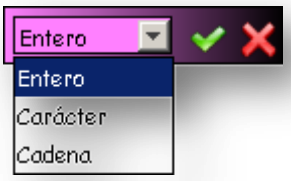


Figura 42 – Ventana donde el usuario elige el tipo de la estructura

Una vez que se ha ejecutado la operación, se deshabilitará su botón, ya que no se podrá crear la cola por segunda vez.

- **Pedir vez**: para que esta operación pueda estar disponible será necesario que la cola se haya creado con anterioridad. Al seleccionar esta operación podrá verse de manera animada cómo se introduce un nuevo elemento en la cola.

  Es el usuario el que indica el dato que se introduce, siendo éste del tipo que se indicó al crear la cola. Para ello, aparecerá una ventana, como la que se muestra en la **figura 43**, donde se podrá introducir el dato. Solo se permite introducir los caracteres que formen datos del tipo indicado en la operación crear, por ejemplo, si al crear se indico que los datos fuesen enteros solo se permiten introducir los caracteres del 0 al 9 y "–". Además, por razones de espacio en el elemento gráfico, solo se permiten, como máximo, elementos de longitud 3.

Figura 43 – Ventana donde el usuario introduce los datos de entrada de la estructura de datos

  Por limitaciones del espacio de la pantalla, hemos considerado que, en cada una de las vistas de las colas haya un máximo de 40 elementos, siendo este un tamaño suficientemente razonable, como para poder entender el comportamiento de las colas. Entonces, cuando se inserte el elemento número 40, el botón de la operación pedir vez se deshabilitará.

- **Avanzar**: al seleccionar esta operación podrá verse de manera animada cómo se elimina el primer elemento de la cola. Para que el botón de esta operación esté habilitado es necesario que se haya creado la cola y que contenga algún elemento.
- **Primero**: al seleccionar esta operación se devolverá el valor del primer elemento de la cola, y se señalará de forma visual el elemento. Para que ésta operación esté disponible será necesario que la cola se haya creado y que contenga algún elemento.
- **Es vacía?**: si se selecciona esta operación, la aplicación indicará si la cola está vacía o si contiene algún elemento. La cola estará vacía nada mas crearla o

cuando se hayan atendido (eliminado) todos sus elementos. Para que el botón de esta operación esté habilitado es necesario que se haya creado la cola.

A continuación se muestra como se representa gráficamente las colas para cada una de las visualizaciones y las animaciones que se realizan.

3.3.1.1.- Visualización "Modo Usuario"

En esta visualización queremos mostrar la filosofía de las colas, es decir, que los elementos se van a ir insertando a continuación del último que está en la cola, y no en otro orden, y que solo es accesible el primer elemento de esta. Para ello:

- **Crear:** al seleccionar esta operación aparecerá la cola vacía, representada mediante un conducto, como el que se muestra en la **figura 44**. Se puede observar que este conducto está diseñado con dos aperturas, una de entada y otra de salida.

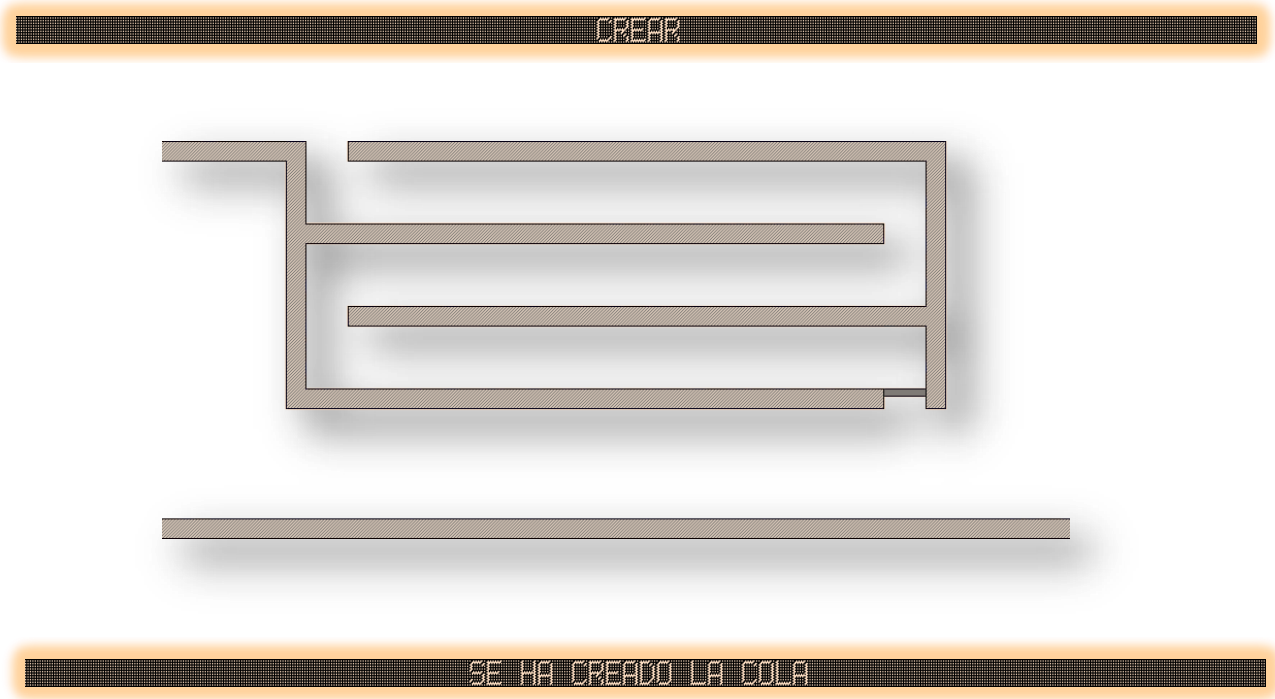
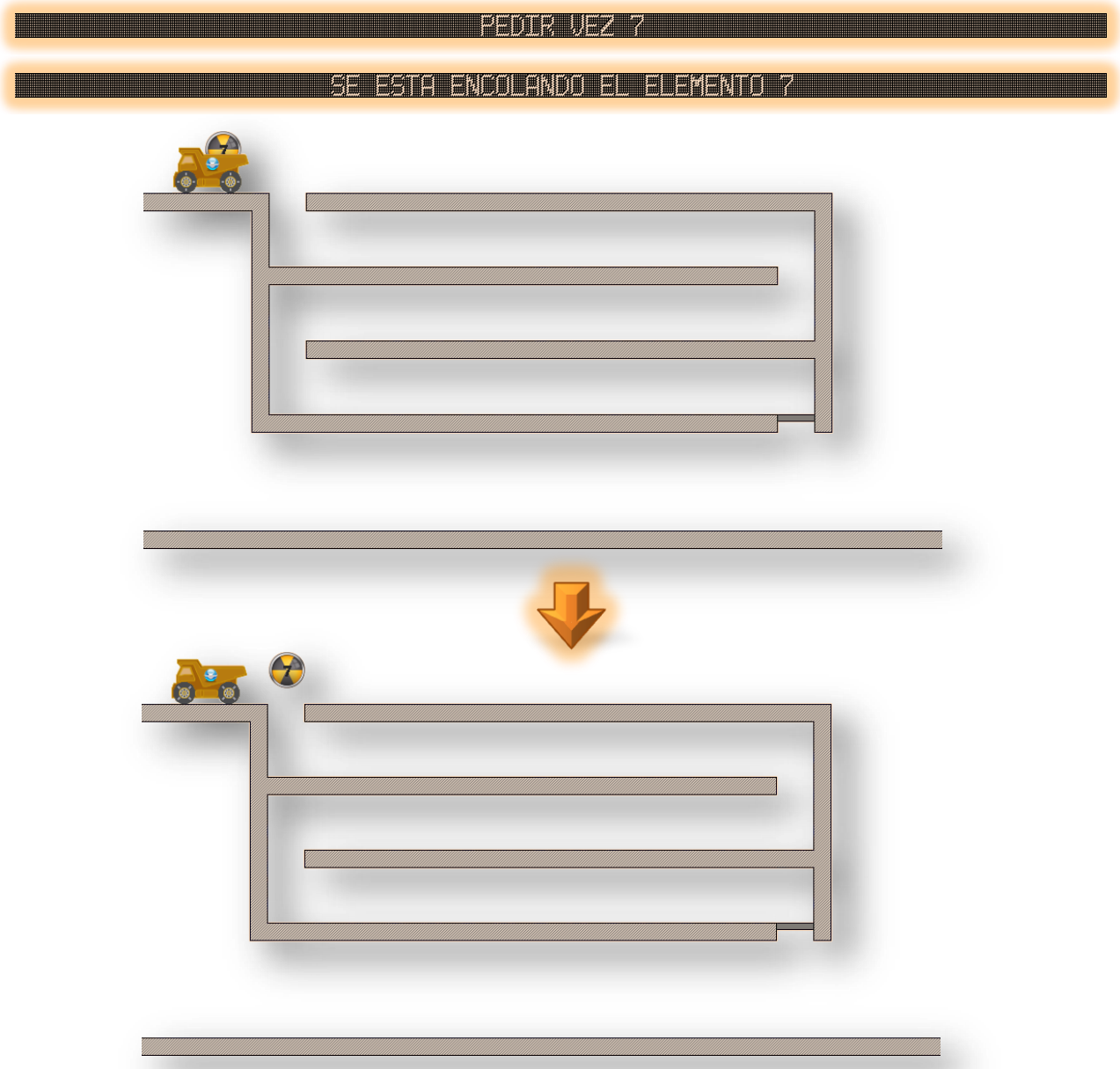


Figura 44 – Animación de la operación Crear. Cola. Visualización de Usuario.

- **Pedir vez:** al seleccionar esta operación, y después de haber indicado el dato que queremos introducir, podrá verse como aparece, en la parte superior izquierda del panel de animación, el nuevo elemento transportado por un camión. El elemento caerá dentro del conducto y se desplazará rodando hasta que se choque con el final de este o con el último elemento. A este elemento no se podrá acceder hasta que no llegue a la compuerta. En la **figura 45** se muestra la secuencia que se produce al pulsar la operación pedir vez.



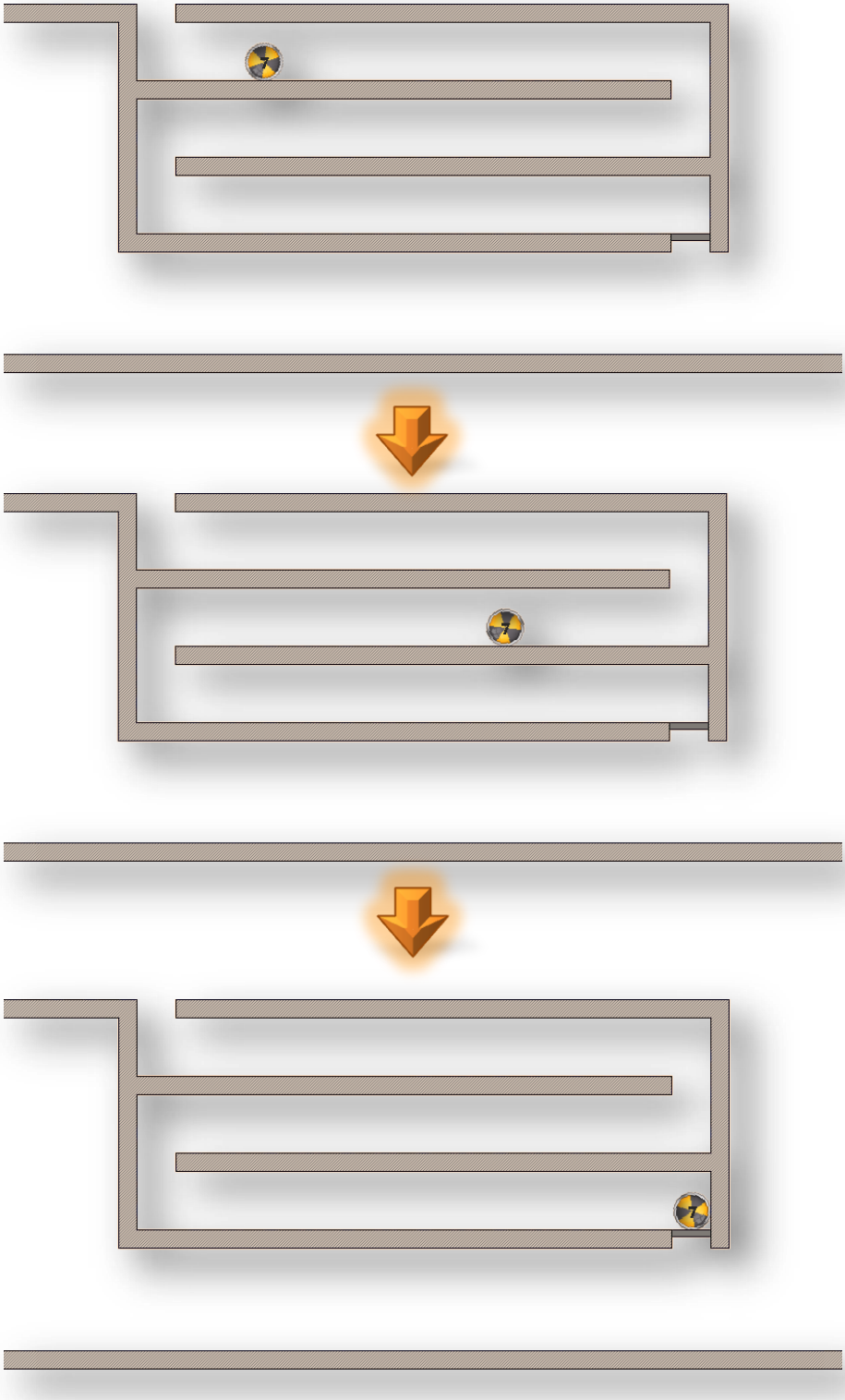


Figura 45 – *Secuencia de animaciones al encolar el elemento 7. Cola. Visualización de Usuario.*

Para continuar la explicación del resto de operaciones de la cola, se han encolado los siguientes elementos en orden:

86, 18, 37, 42, 67, 24, 46, 2, 96,
79 55, 87, 63, 15, 13, 8, 65, 9

Después de encolarlos, la cola queda como se muestra en la **figura 46**.

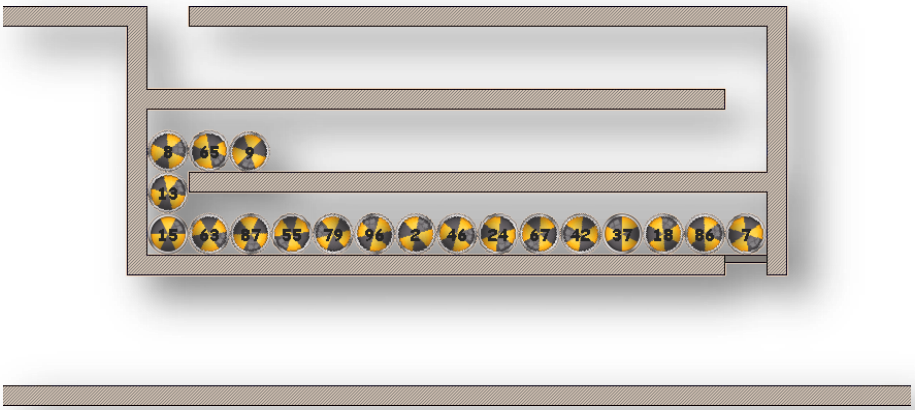
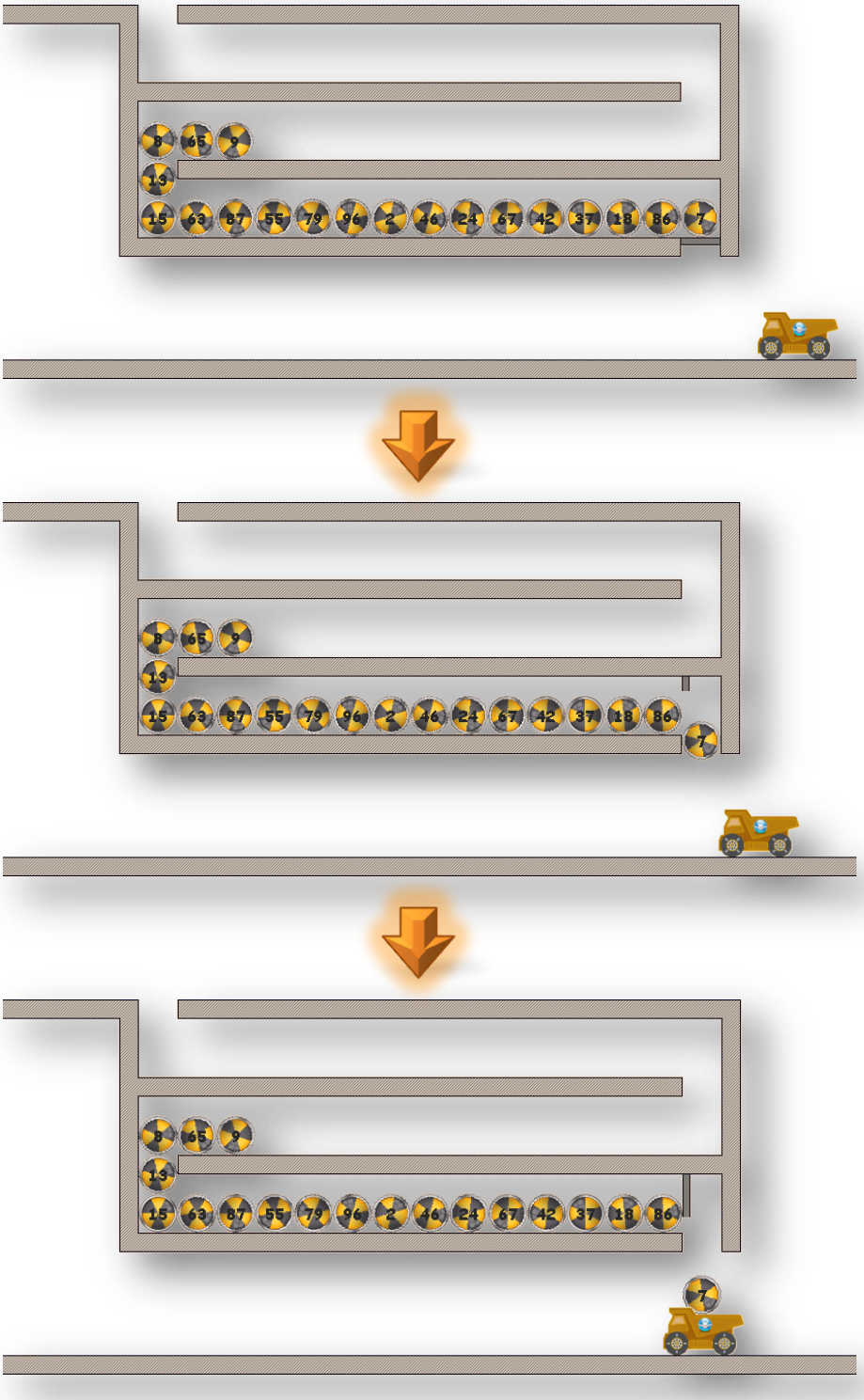


Figura 46 – Encolamos varios elementos. Cola. Visualización de Usuario

- **Avanzar**: al seleccionar esta operación se desea procesar un elemento, para ello, como por la parte superior es imposible sacar ningún elemento ya que la ley de gravedad nos lo impide, se activa el mecanismo que abre la compuerta para que el primer elemento caiga al camión que hemos mandado a recogerlo. La operación de avanzar se detalla en la **figura 47**.

AVANZAR

SE ESTA DESENCOLANDO EL PRIMER ELEMENTO DE LA COLA, EL ELEMENTO 7



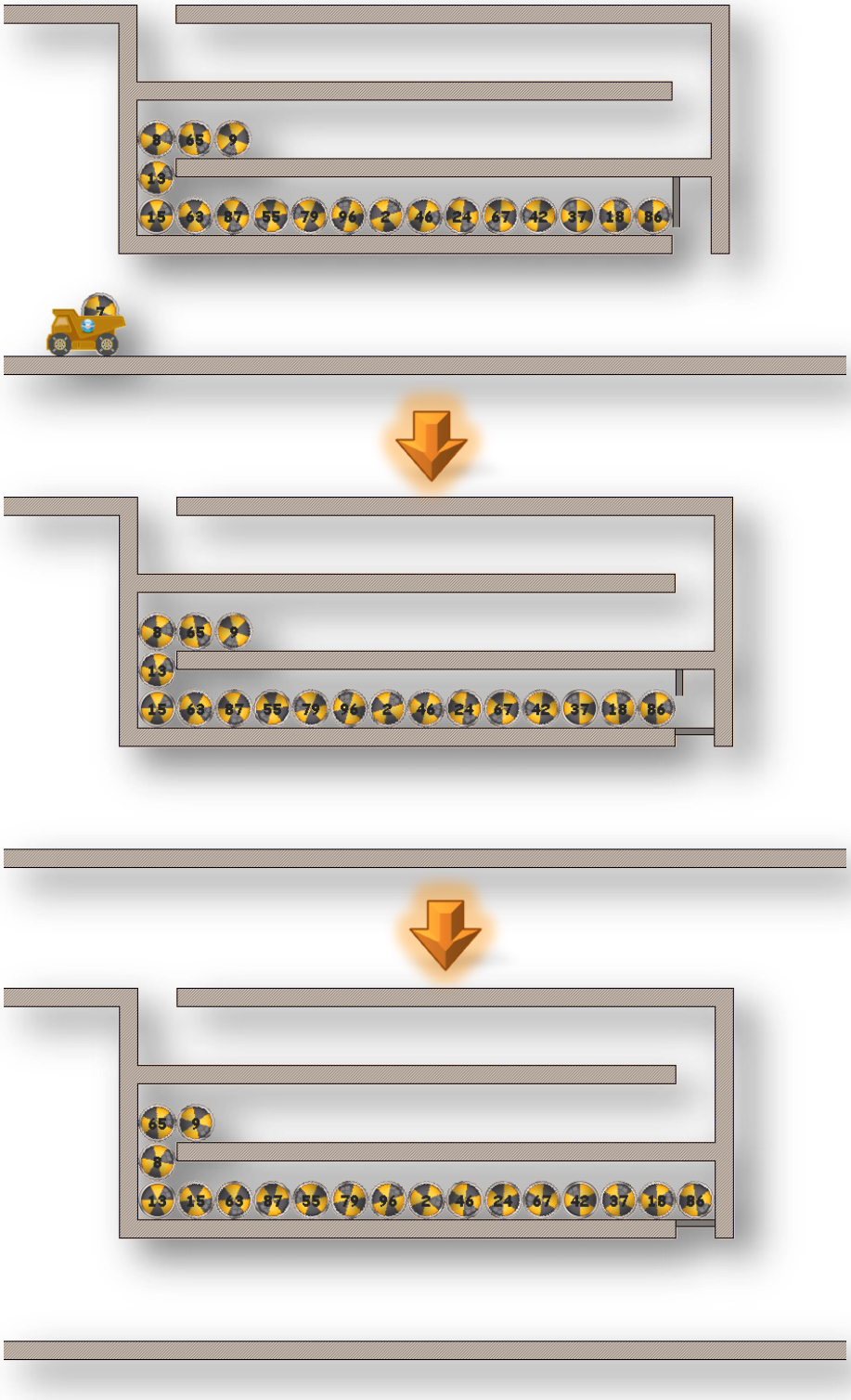


Figura 47 – Animación de la operación Avanzar. Cola. Visualización de Usuario.

- **Primero**: al seleccionar esta operación aparecen 4 flechas que señalan al primer elemento de la cola (ver **figura 48**)

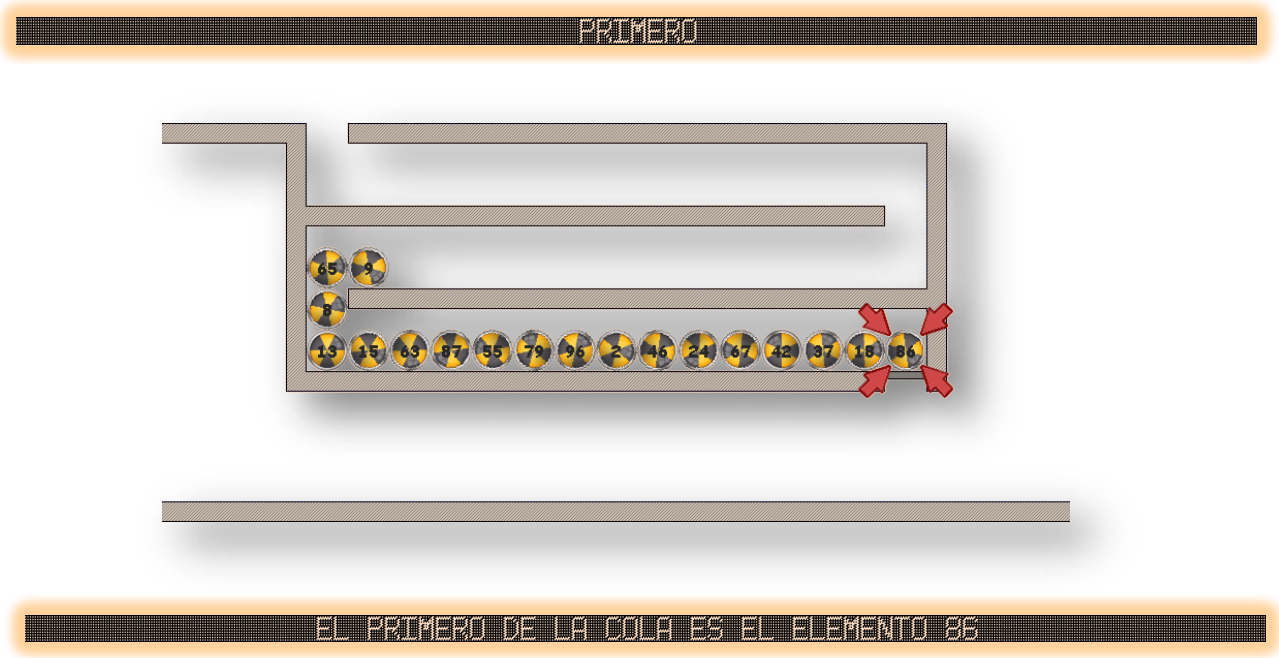


Figura 48 – Animación de la operación Primero. Cola. Visualización de Usuario.

- **Es vacía?**: *al seleccionar esta operación se muestra en el cuadro de dialogo si la cola está o no vacía (ver* ***figuras 49*** *y* ***50****).*

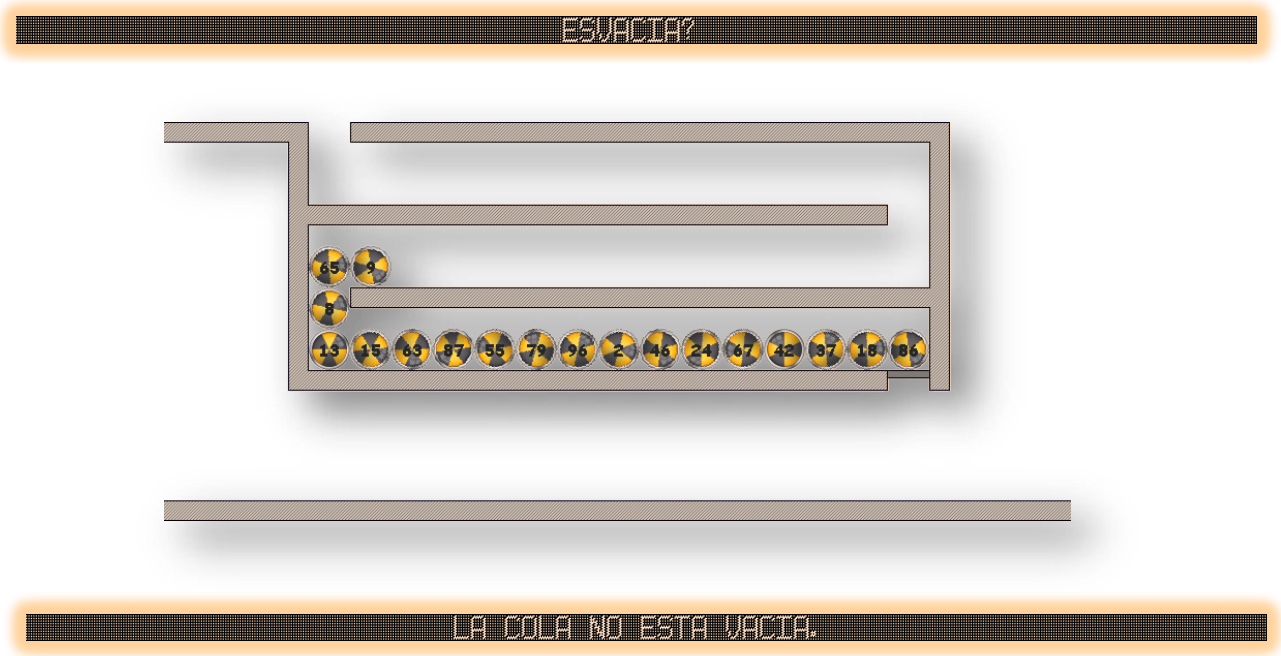


Figura 49 – Animación de la operación EsVacia?. Cola no vacía. Visualización de Usuario.

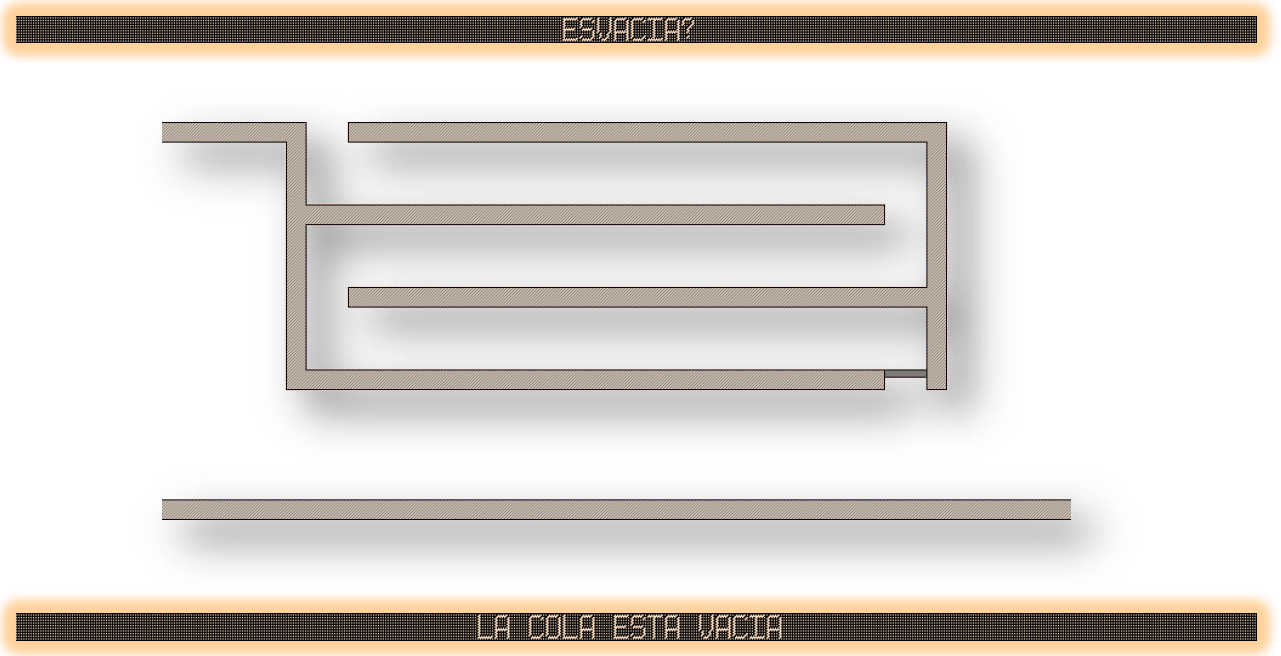


Figura 50 – Animación de la operación EsVacia?. Cola vacía. Visualización de Usuario.

3.3.1.2.- Visualización "Modo Implementación Estática"

En esta visualización se mostrará la representación estática de las colas utilizando un vector circular. La idea es utilizar un vector donde vamos colocando los elementos de la pila, en orden creciente de posiciones según se van añadiendo, y dos índices que apuntarán al comienzo (primero) y final (último). Gestionamos el vector de forma circular, para no tener que desplazar los elementos cuando el vector esté ocupado hasta la última posición pero haya posiciones libres al principio del mismo. Para cada posición en el vector se define su siguiente mediante la función:

$$sig(i) = \begin{cases} i+1 \; si \; 1 \leq i < N \\ 1 \quad si \quad i = N \end{cases} = (i \; mod \; N) + 1$$

En el caso de vacío necesitamos inicializar los índices primero y último, para ello primero = 1 y último = N. De esta forma $sig(último) = primero$ cuando la cola está vacía y cuando la cola está llena. Para distinguir estas situaciones se necesita almacenar otro valor que contenga el tamaño de la cola (tamaño).

Esta representación tiene el inconveniente, al igual que en la pila estática (**sección 3.3.1.2**), de que el tamaño de la estructura queda restringido por la capacidad del soporte de almacenamiento, en este caso por el tamaño del vector.

- ***Crear***: al seleccionar está operación aparecerá la estructura que representa el vector circular de 40 elementos, los índices primero y último inicializados a 1 y 40, respectivamente, y tamaño inicializado a 0, ya que la pila se crea vacía.
  Al realizar la operación crear sobre una cola representada estáticamente, se tiene que crear un vector, de forma que se reserva un espacio en memoria correspondiente al tamaño del vector. En la **figura 51**, se muestra el vector que representa la cola con forma circular, esto es así ya que hemos creído que resultaría más fácil de entender está implementación de las colas.

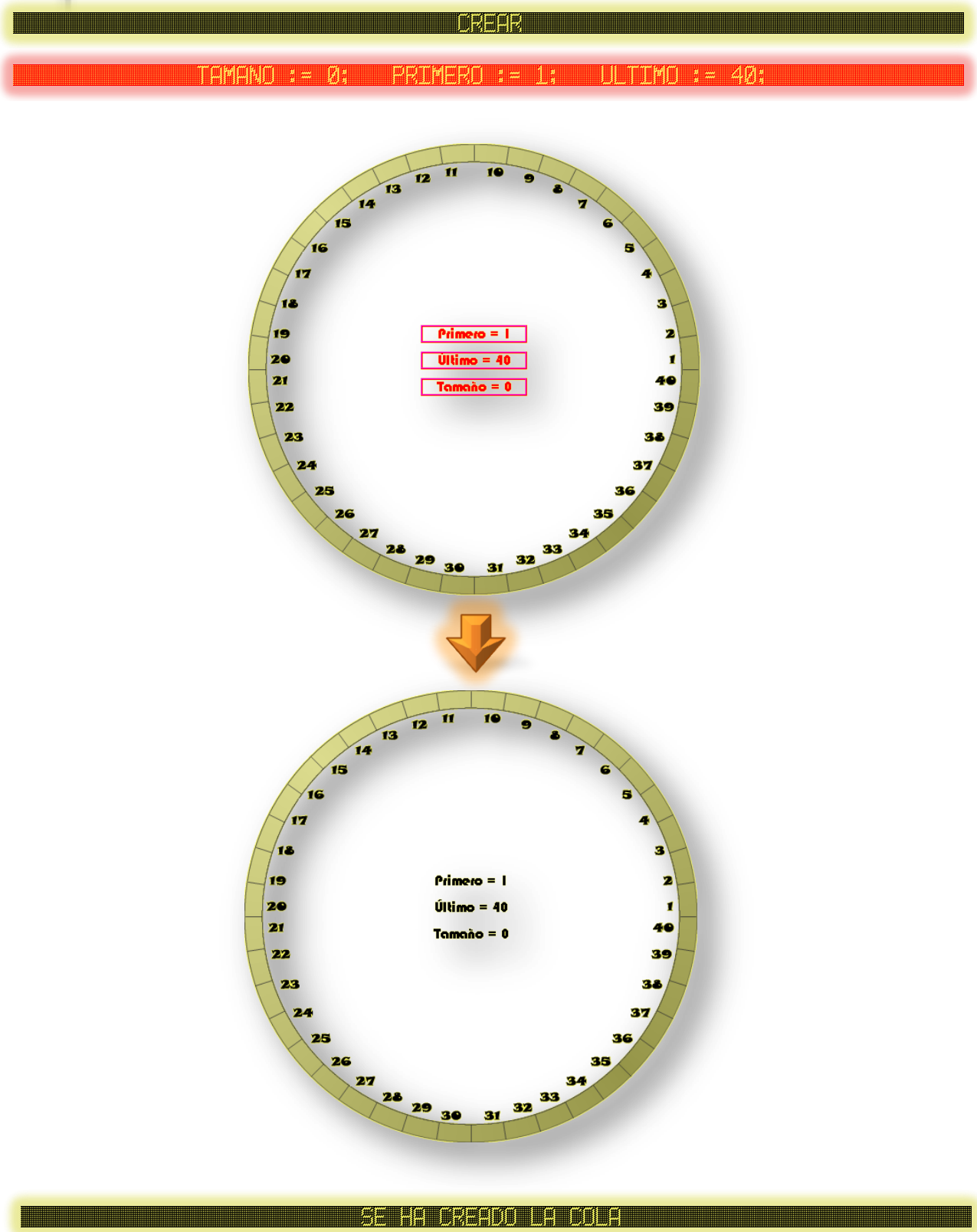
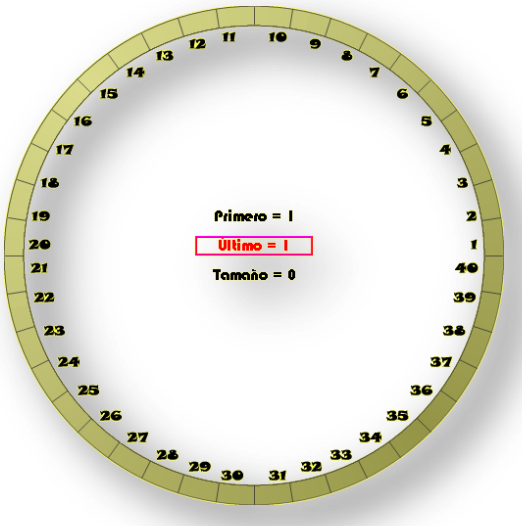


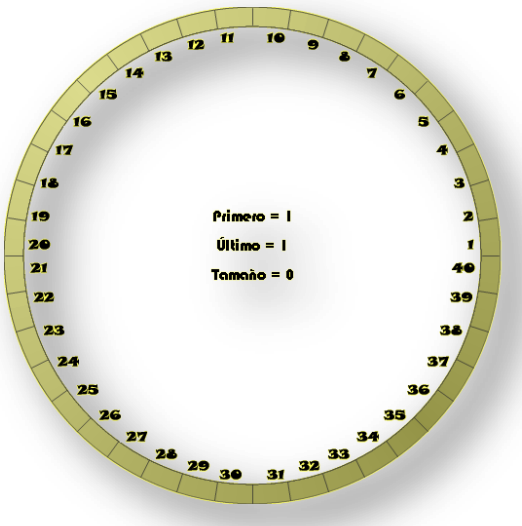
Figura 51 – *Animación de la operación Crear. Cola. Visualización Estática.*

- **Pedir vez**: al seleccionar esta operación, y después de haber indicado el dato que queremos introducir, se mostrará de forma animada cómo se introduce un nuevo elemento en la cola. Para ello, se aumenta el índice último haciendo $(último \bmod N) + 1$, se aumenta el tamaño de la cola en una unidad y se introduce el elemento en dicha posición del vector. (ver la secuencia de la ***figura* 52**)

```
PEDIR VEZ ?
ULTIMO := (ULTIMO MOD 40) + 1;
```




```
CONTENIDO(ULTIMO) := ?;
```



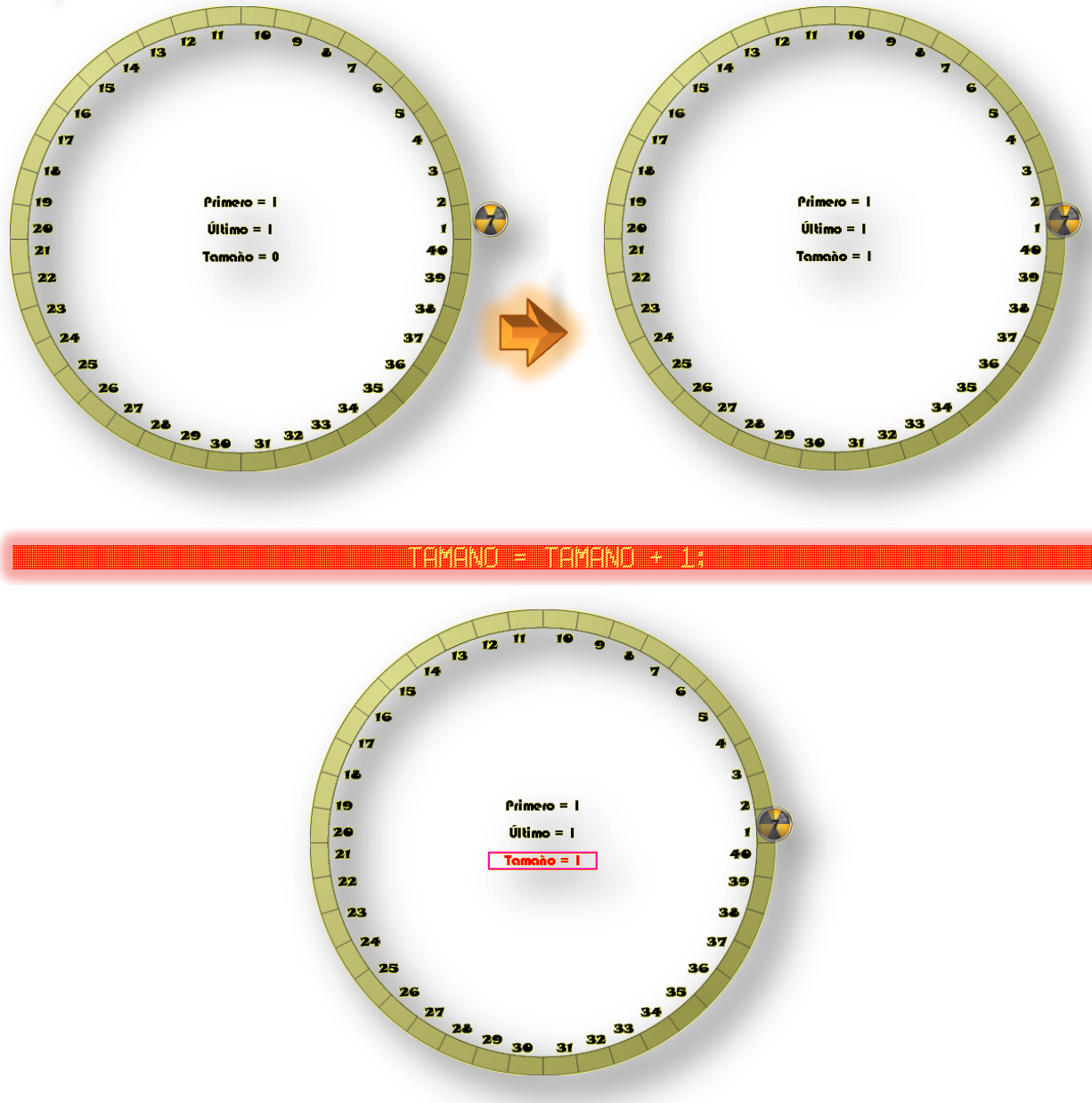


Figura 52 – *Animación de la operación PedirVez el elemento 7. Cola. Visualización Estática.*

Para continuar la explicación del resto de operaciones de la cola, se han encolado los siguientes elementos en orden:

86, 18, 37, 42, 67, 24, 46, 2, 96,
79 55, 87, 63, 15, 13, 8, 65, 9

Después de encolarlos, la cola queda como se muestra en la **figura 53**.

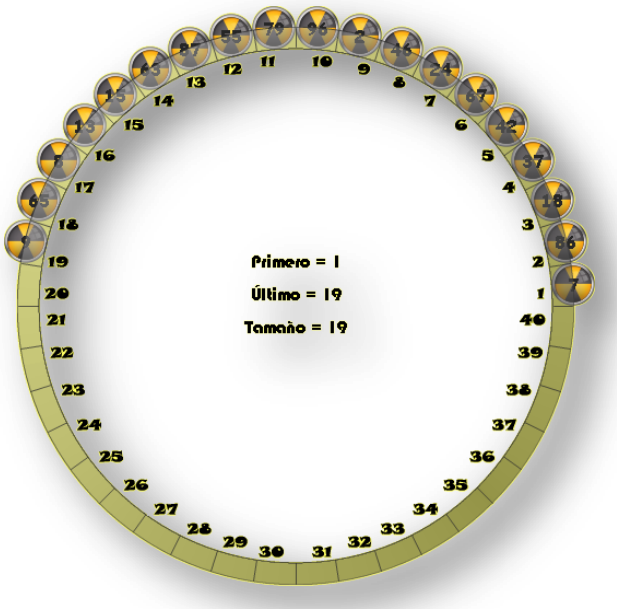
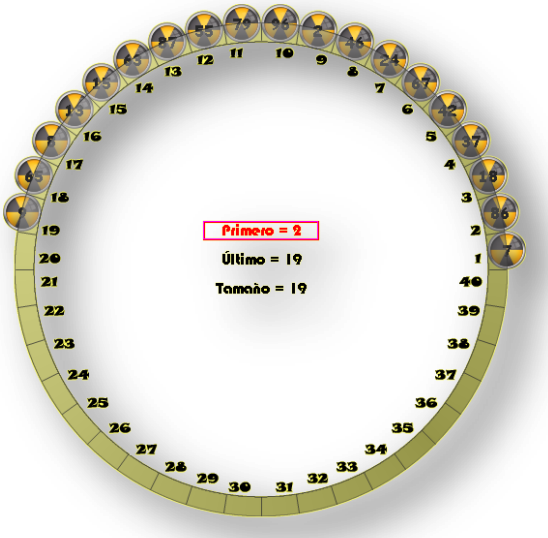


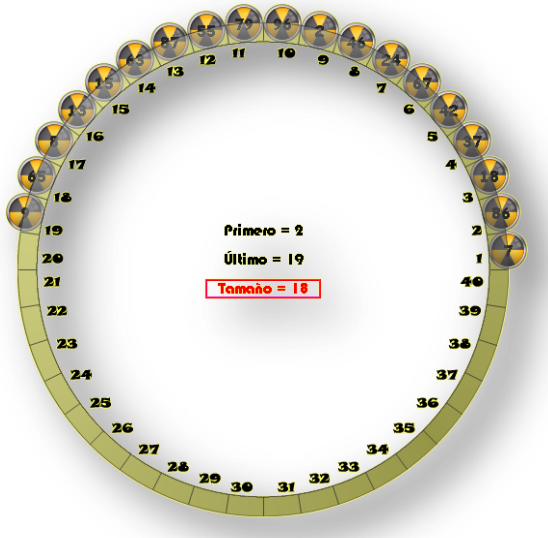
Figura 53 – Encolamos varios elementos. Cola. Visualización Estática

- ***Avanzar***: *al seleccionar esta operación, se aumenta el índice primero haciendo* $(primero \bmod N) + 1$, *apuntando al nuevo primero de la cola y se disminuye en una unidad su tamaño. Para una mayor claridad eliminamos dicho elemento del vector, aunque realmente se quede como basura dentro de él, (ver secuencia de la* ***figura 54***).

```
AVANZAR
PRIMERO := (PRIMERO MOD 40) + 1;
```




```
TAMANO := TAMANO - 1;
```



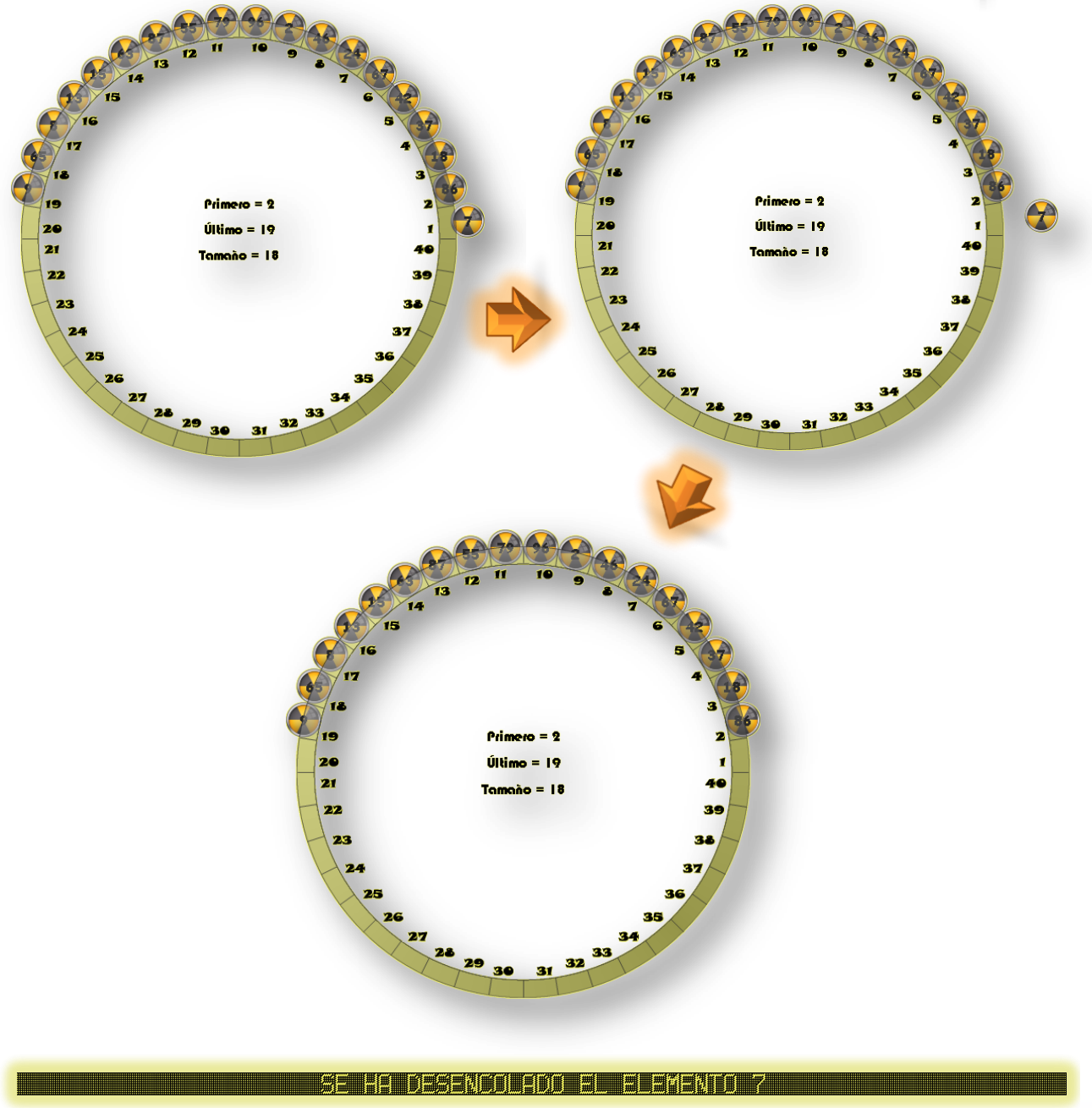


Figura 54 – *Animación de la operación Avanzar el elemento 7. Cola. Visualización Estática.*

- **Primero**: al seleccionar esta operación aparecen 4 flechas que señalan el elemento que está contenido en la posición primero del vector, (ver **figura 55**).

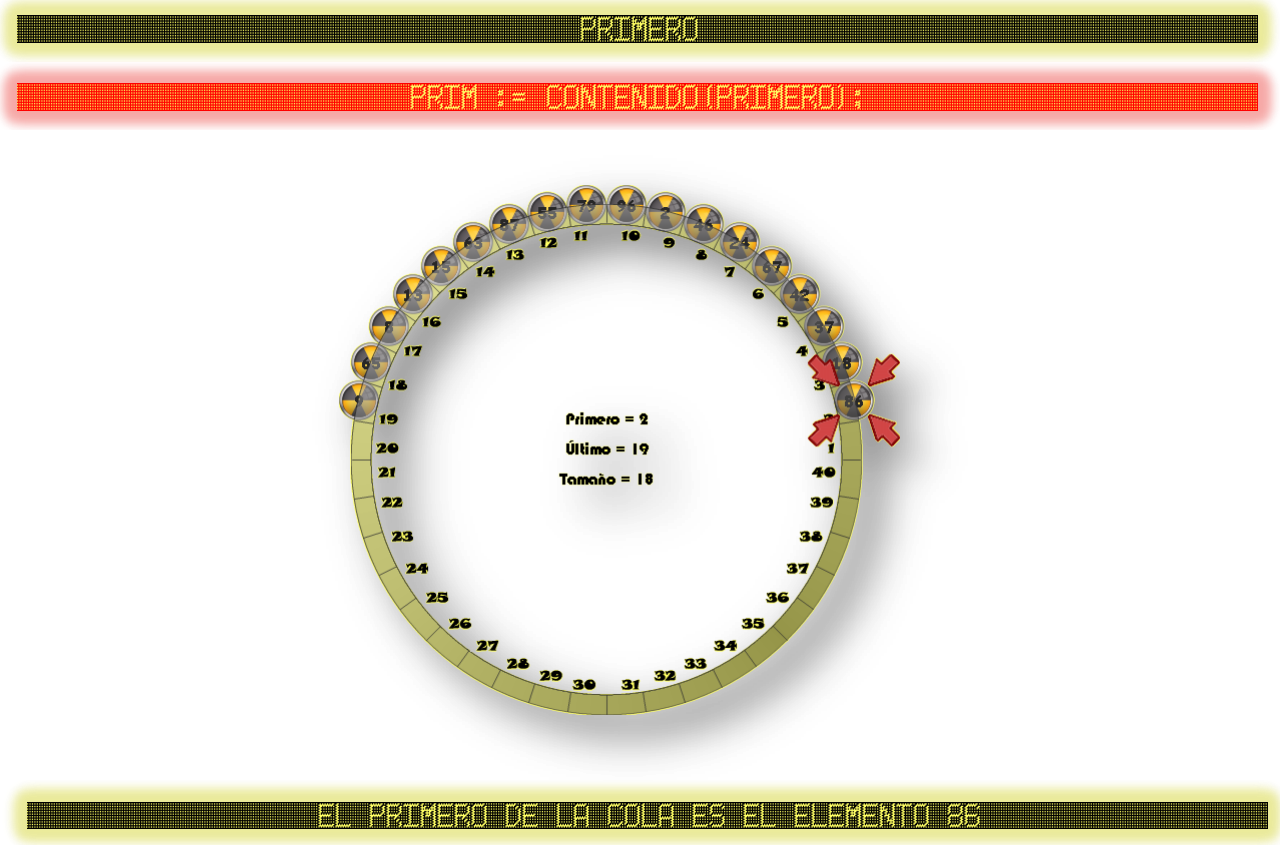
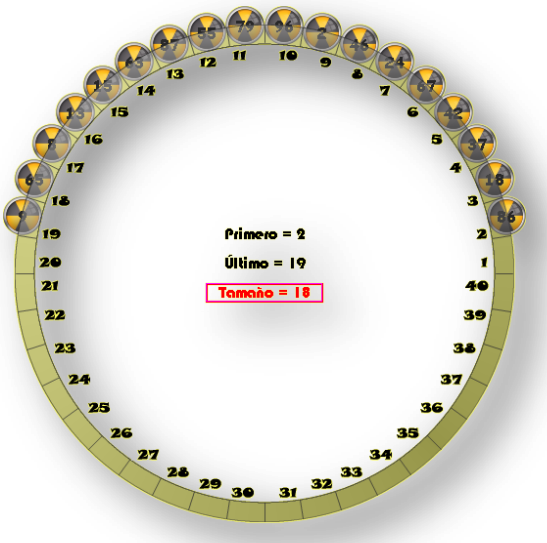


Figura 55 – Animación de la operación Primero. Cola. Visualización de Usuario.

- **Es vacía?**: al seleccionar esta operación se muestra en el cuadro de dialogo si la pila está o no vacía, es decir, si tamaño es igual a cero o distinto de cero, respectivamente, (ver **figuras 56** y **57**).

ESVACIA?

ESVACIA? := (TAMANO == 0);

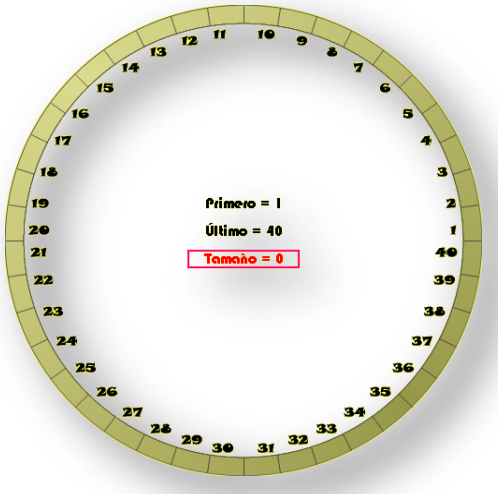


LA COLA NO ESTA VACIA

Figura 56 – *Animación de la operación EsVacia?. Cola no vacía. Visualización de Usuario.*

ESVACIA?

ESVACIA? := (TAMANO == 0);



LA COLA ESTA VACIA

Figura 57 – *Animación de la operación EsVacia?. Cola vacía. Visualización de Usuario.*

3.3.1.3.- Visualización "Modo Implementación Dinámica"

En esta visualización se mostrará la representación dinámica de las colas utilizando punteros. Esta representación se realiza mediante una estructura lineal enlazada, que mantiene el orden en que los elementos fueron introducidos en la cola. Para que el coste de todas las operaciones sea constante se tiene acceso directo a los extremos de la estructura mediante punteros que señalan al primer elemento (extremo por donde se consultan y eliminan los elementos) y al último (extremo por donde se introducen los elementos).

La cola se mostrará mediante nodos (que representan los elementos) con un puntero o enlace, el cual apunta al nodo siguiente. El primer elemento o comienzo de la cola, estará apuntado por el puntero "primero" y el último elemento de la cola por el puntero "último".

- **Crear**: al seleccionar esta operación los punteros a los estremos no apuntan a ninguna estructura, apuntan a "null", representado como una toma de tierra, (ver **figura 58**).

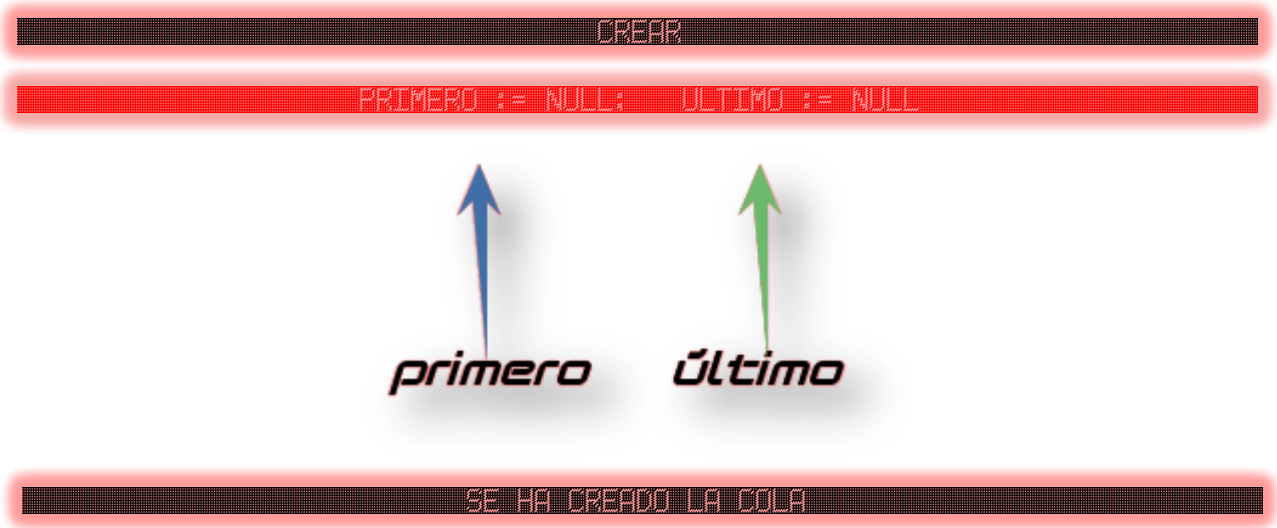
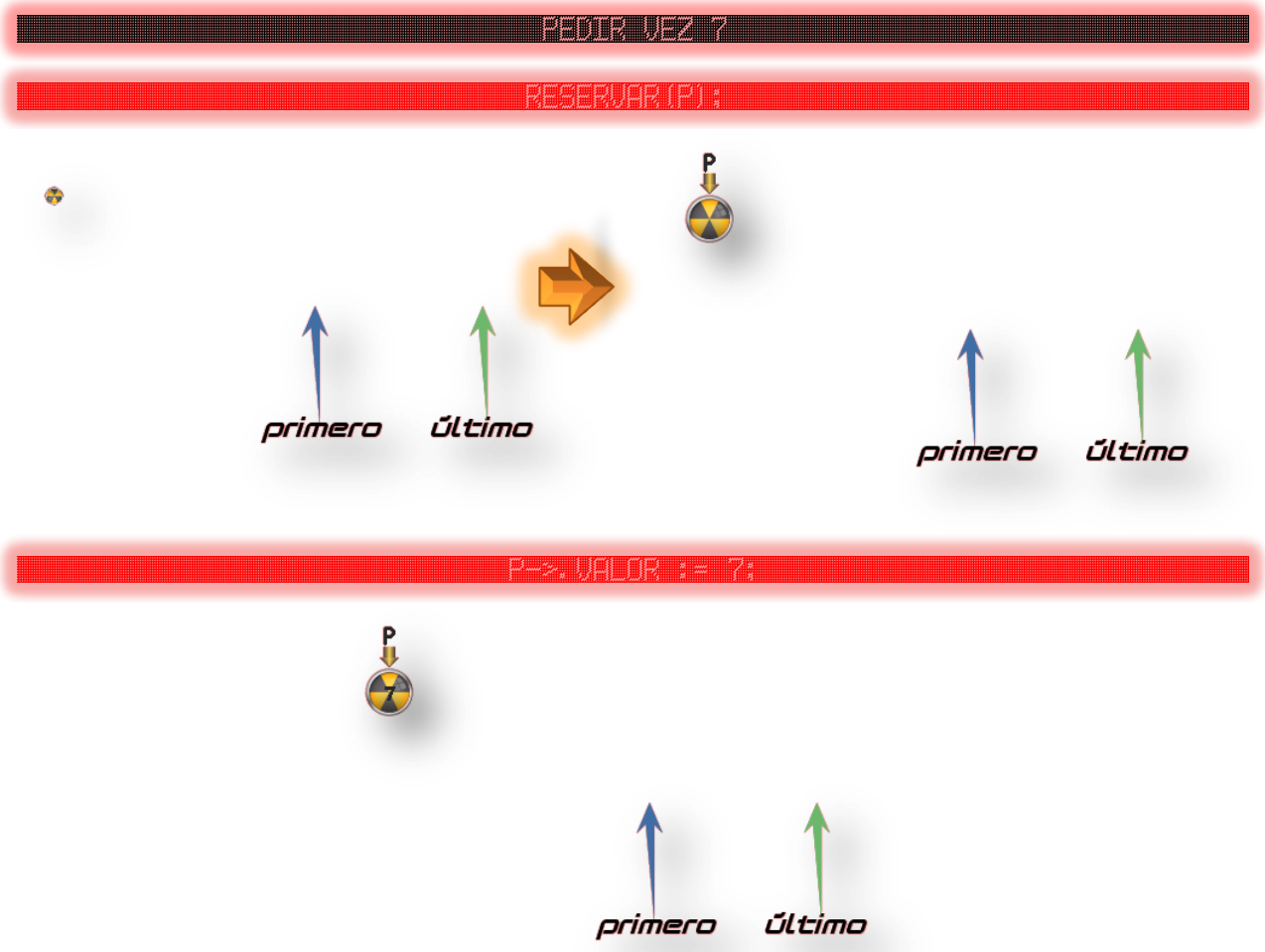


Figura 58 – Animación de la operación Crear. Cola. Visualización Dinámica.

- **Pedir vez**: *al seleccionar esta operación, y después de haber indicado el dato que queremos introducir, se mostrará de forma animada cómo se introduce un nuevo elemento en la cola. Se utiliza un puntero auxiliar* **P** *para hacer la reserva de la memoria dinámica. Como el elemento se introduce por el final de la cola, el nuevo elemento se enlaza con el puntero último. Solo en el caso de que la cola estuviera vacía es necesario modificar el puntero primero. La secuencia de animaciones puede observarse en las* ***figuras* 59** y **60**.



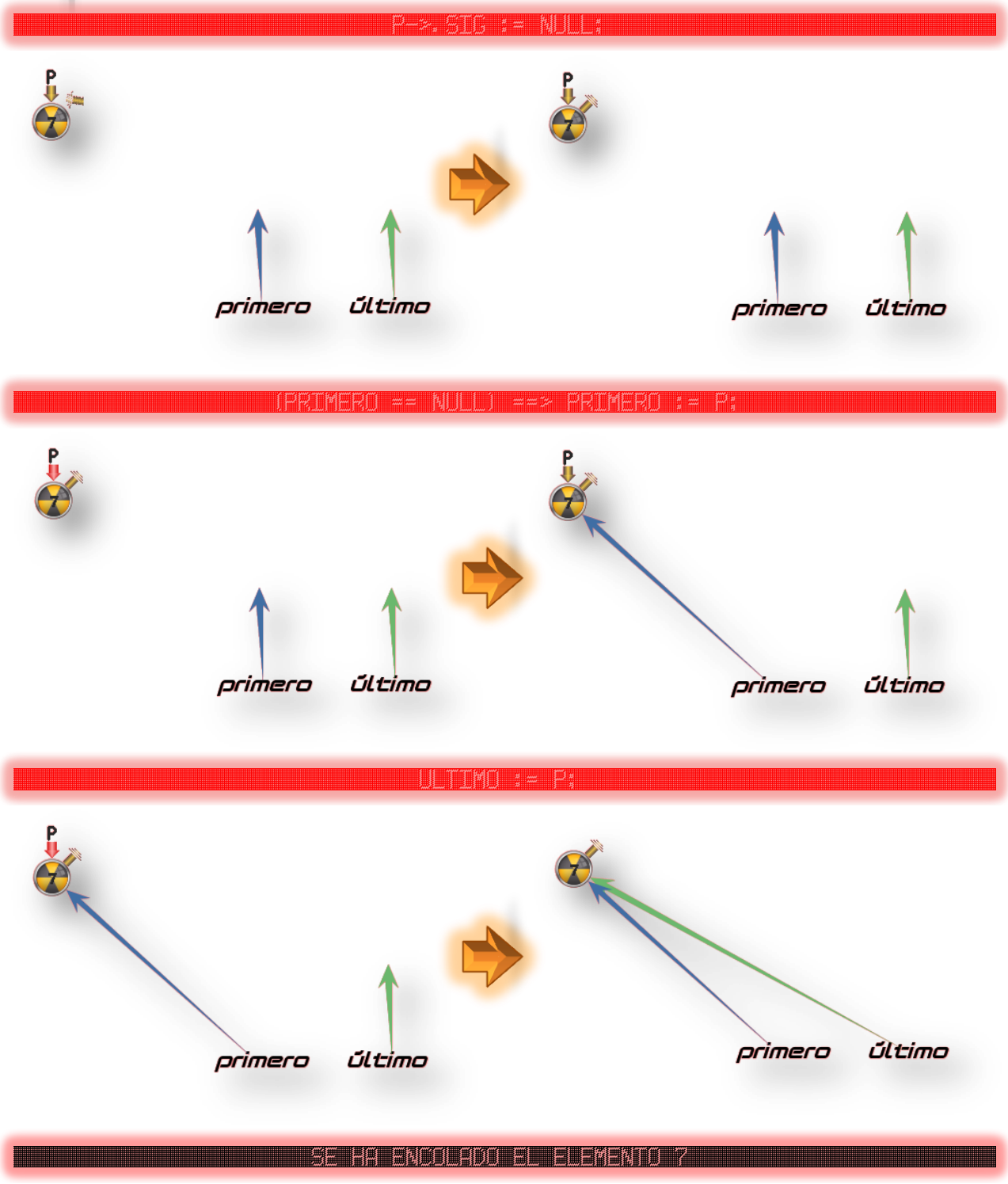
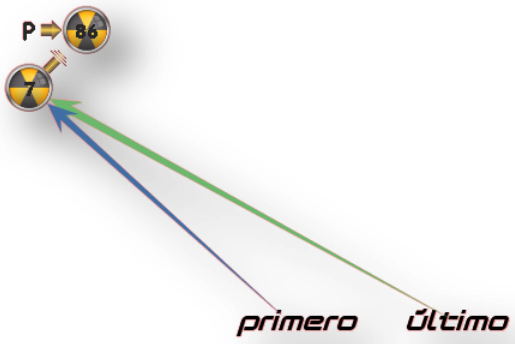


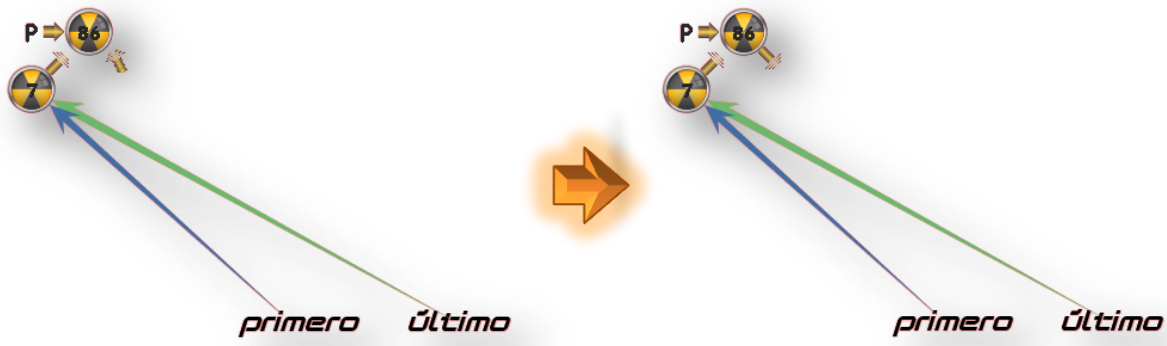
Figura 59 – Animación de la operación PedirVez del elemento 7. Cola. Visualización Dinámica.

```
PEDIR VEZ 86
RESERVA(P);
```


```
P->.VALOR := 86;
```




```
P->.SIG := NULL;
```



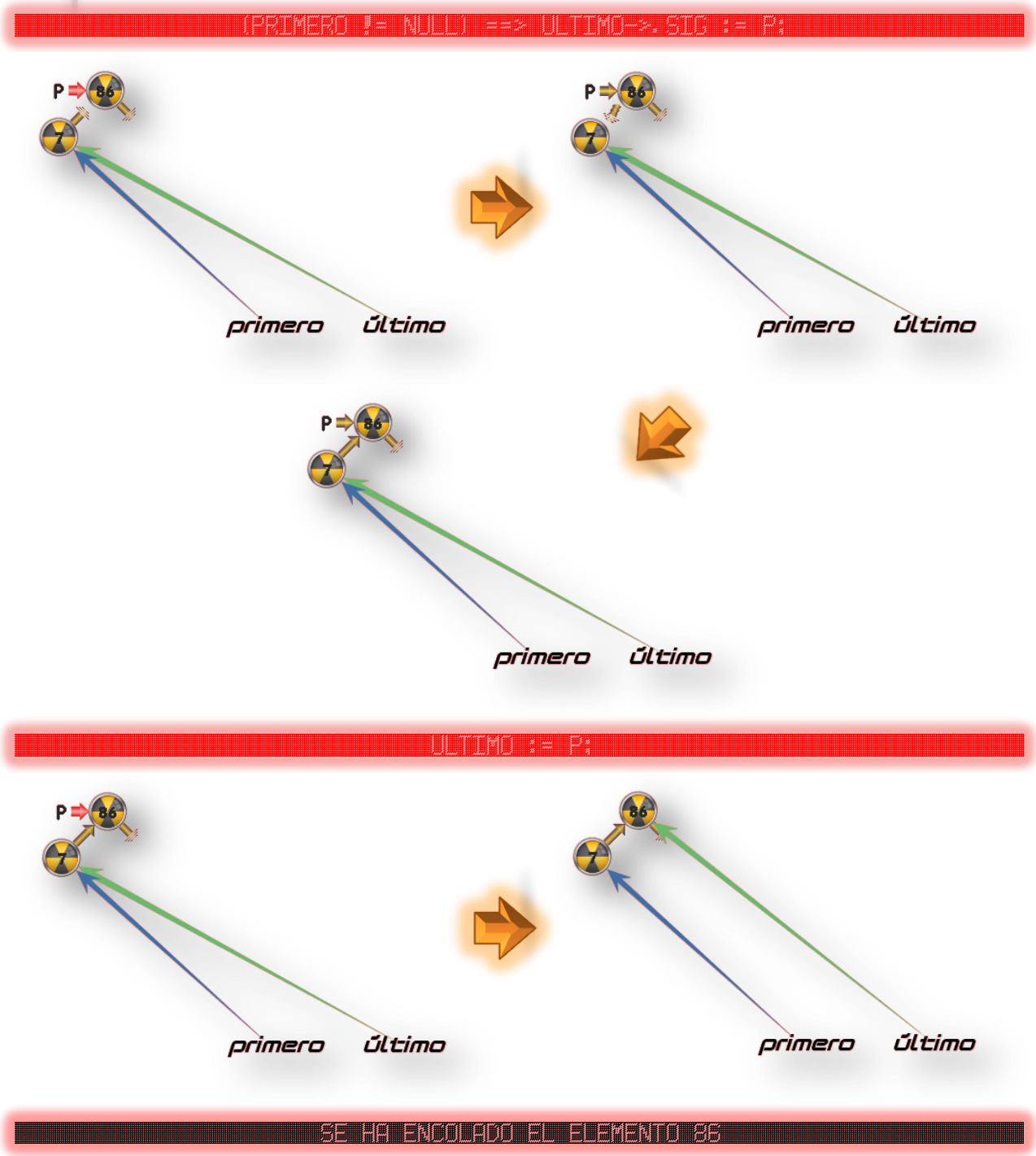
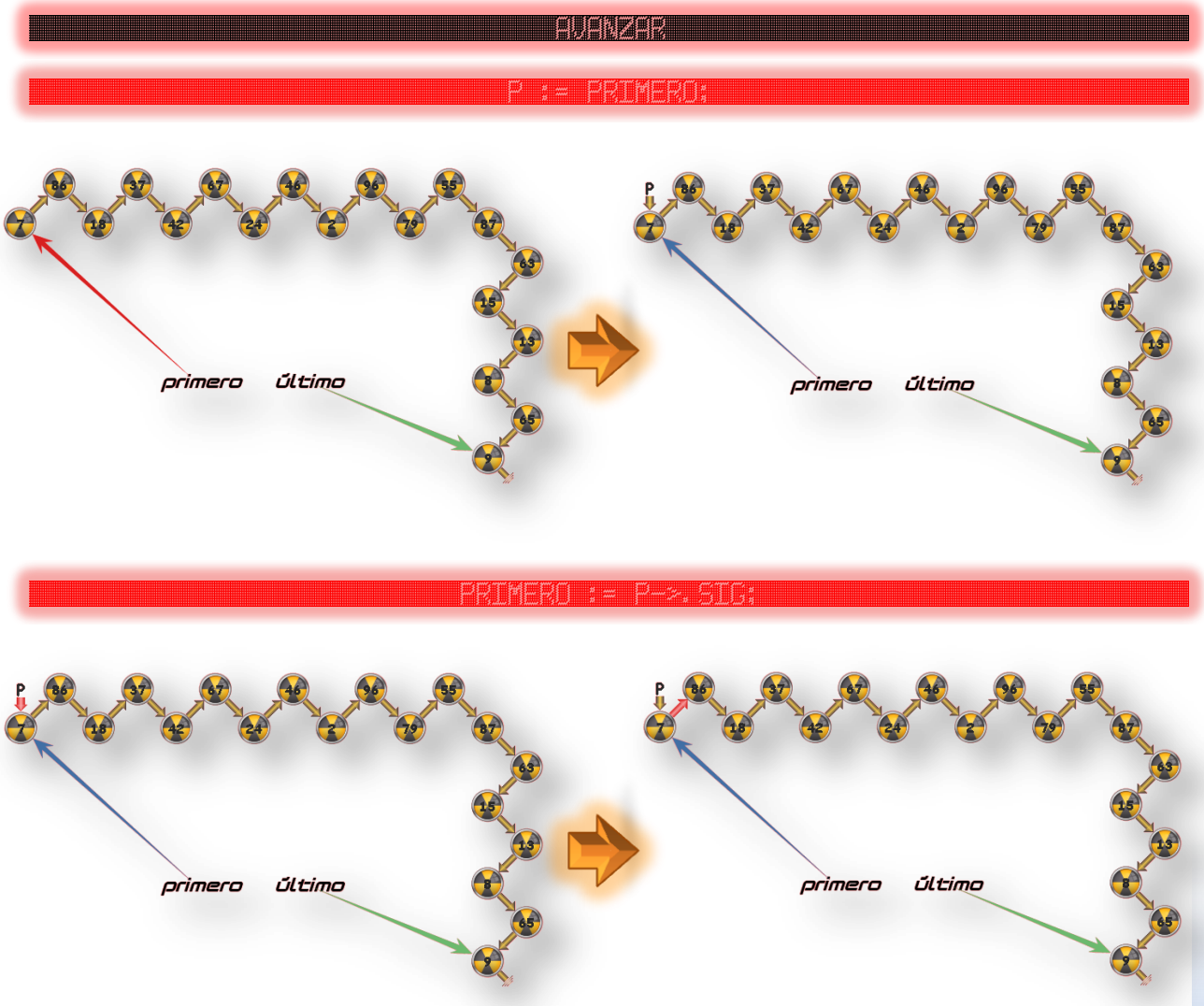


Figura 60 – *Animación de la operación PedirVez del elemento 86. Cola. Visualización Dinámica.*

- **Avanzar**: al seleccionar esta operación podrá verse de manera animada cómo se elimina el primer elemento. Como en pedir vez, también se utiliza un puntero auxiliar **P**, en este caso, para liberar la memoria dinámica asignada al primer elemento. En el caso de que la cola se quede vacía es necesario que el puntero último apunte a **"null"**. La secuencia de animaciones podrá observarse en las ***figuras* 61** y **62**.



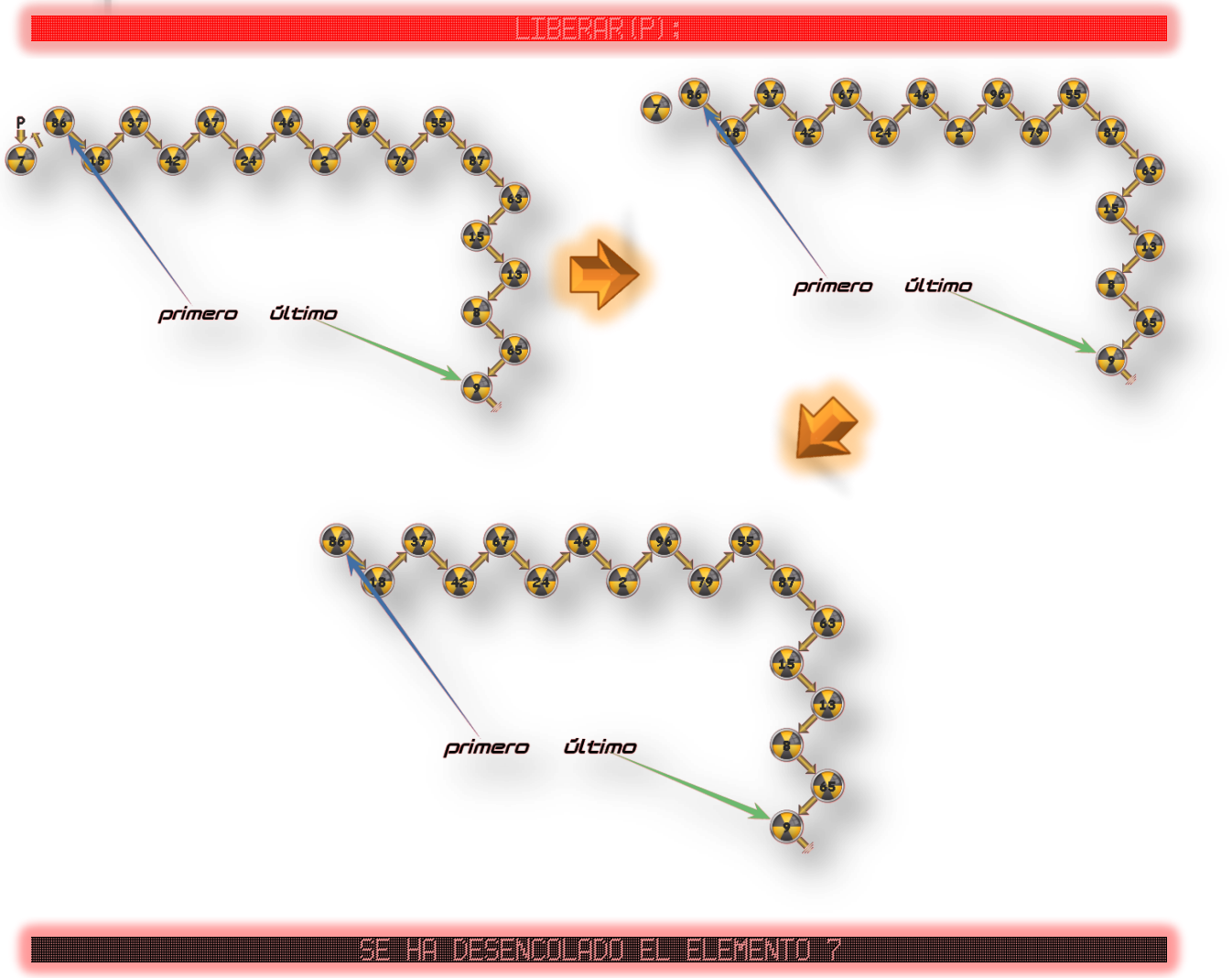
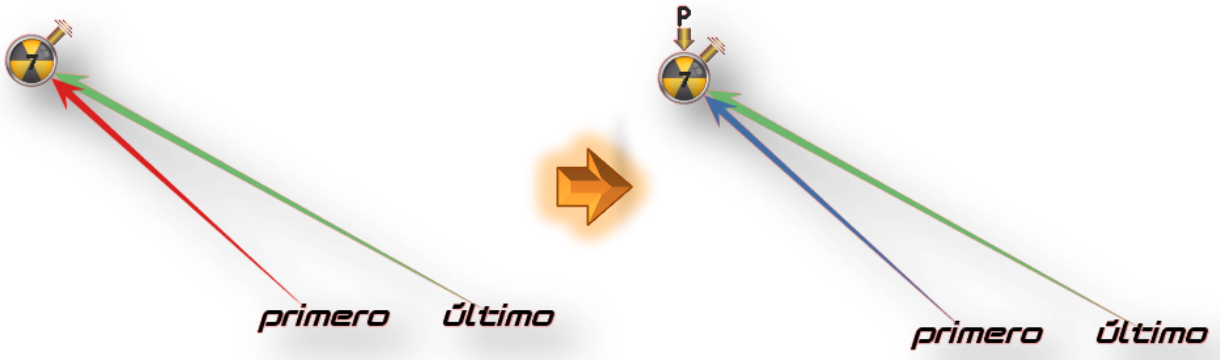


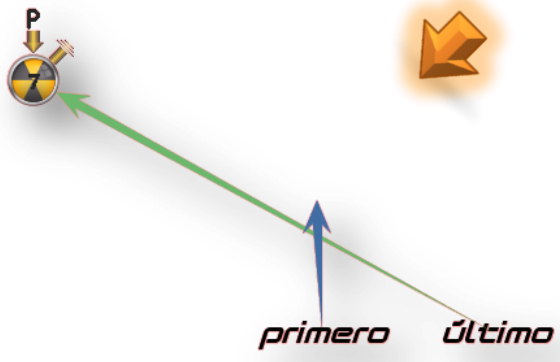
Figura 61 – Animación de la operación Avanzar. Cola. Visualización Dinámica.

AVANZAR

P := PRIMERO;



PRIMERO := P->.SIG;



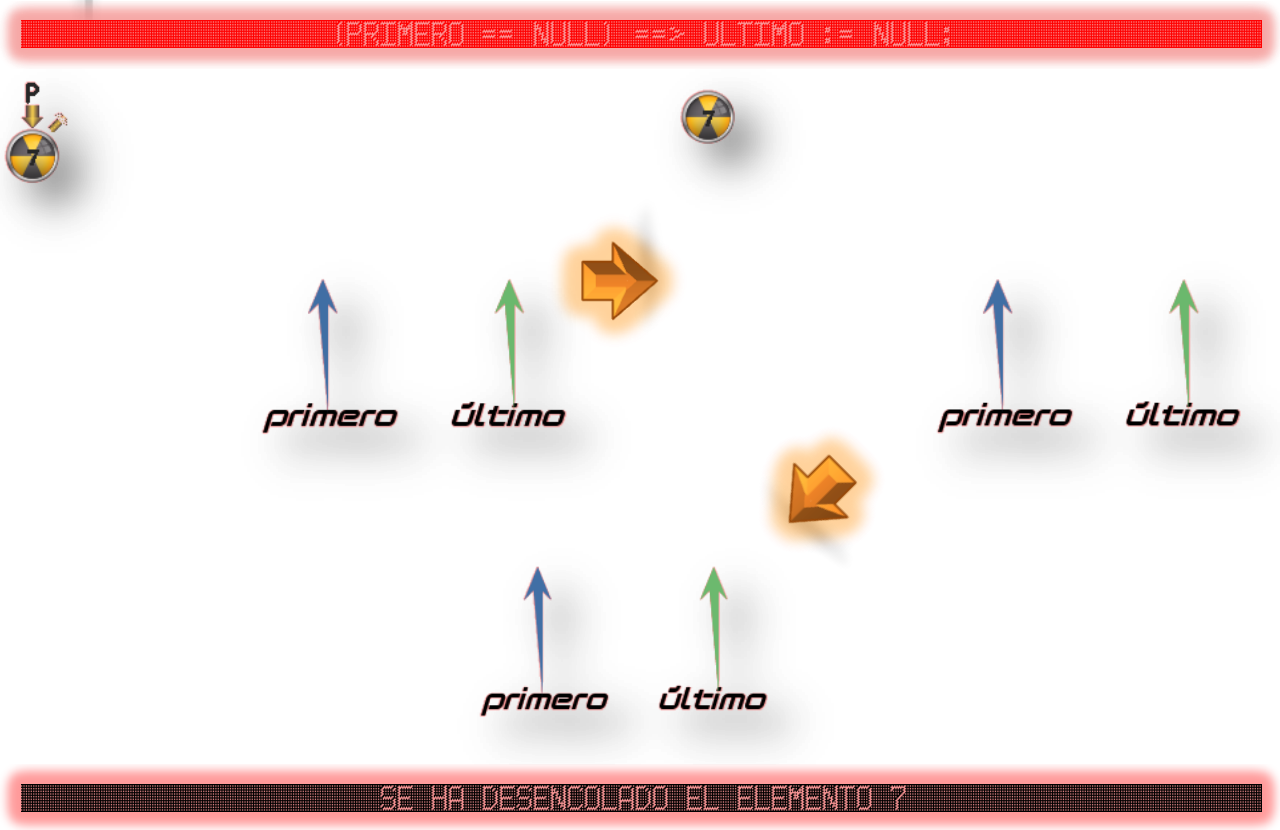


Figura 62 – Animación de la operación Avanzar el elemento 7. Cola. Visualización Dinámica.

- **Primero**: al seleccionar esta operación aparecen 4 flechas que señalan el elemento al que apunta primero, (ver **figura 63**).

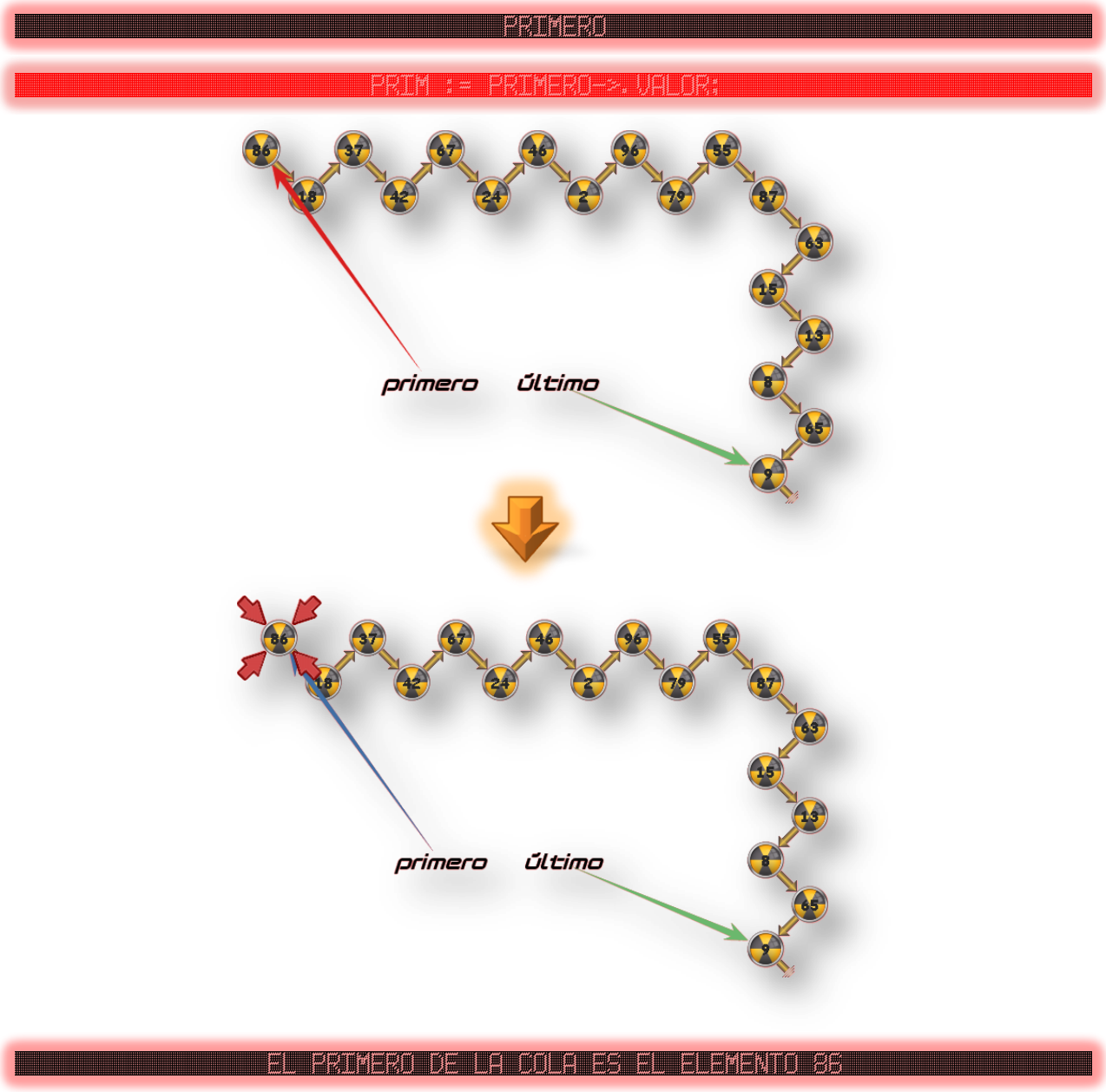


Figura 63 – Animación de la operación Primero. Cola. Visualización Dinámica.

- **Es vacía?**: *al seleccionar esta operación se muestra en el cuadro de dialogo si la pila está o no vacía, es decir, si primero apunta a* **"null"** *o no, respectivamente, (ver* ***figuras*** **64** y **65**).

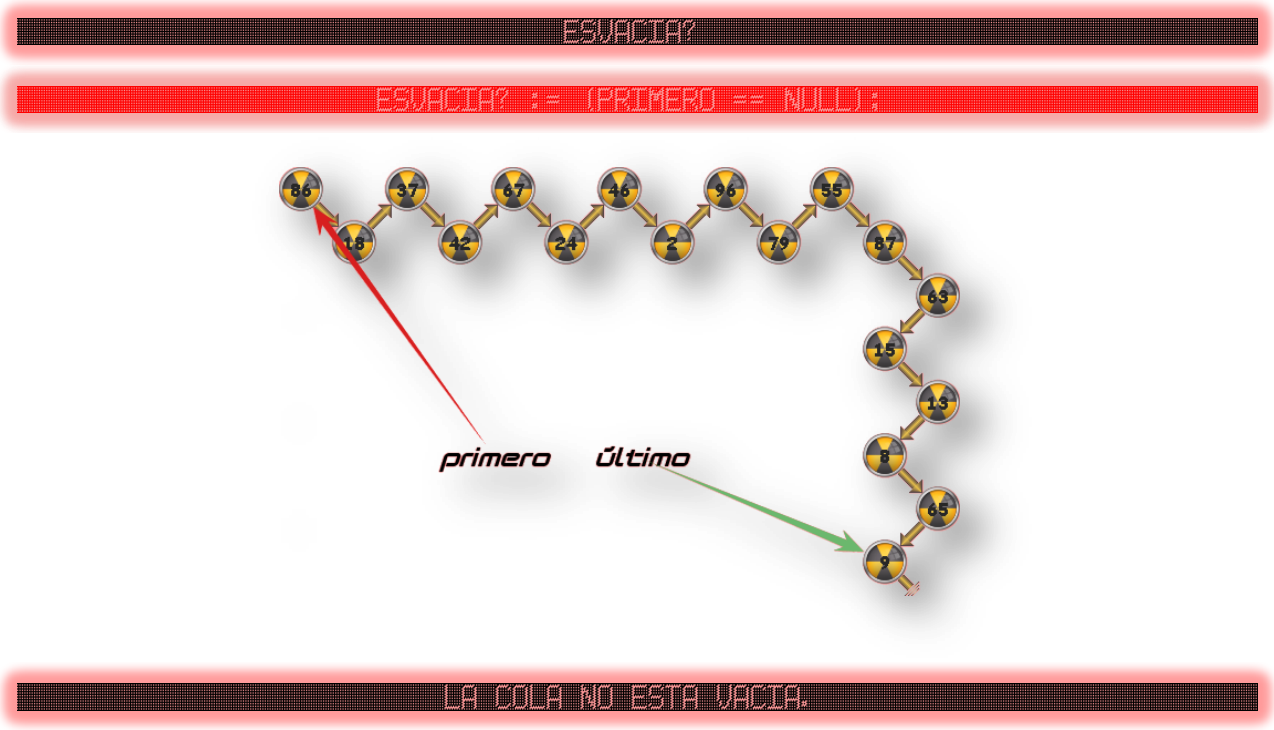


Figura 64 – *Animación de la operación EsVacia?. Cola no vacía. Visualización Dinámica.*

Figura 65 – *Animación de la operación EsVacia?. Cola vacía. Visualización Dinámica.*

3.3.2.- Árboles Binarios de Búsqueda

Las estructuras arbóreas generalizan las estructuras lineales (vistas en las **secciones 3.3.1 y 3.3.2**), de forma que en vez de pasar de un dato al (único) siguiente, tenemos varios siguientes que se consideran hijos del dato.

En los arboles binarios todo nodo tiene siempre dos hijos, aunque uno de ellos o ambos pueden ser vacios. Los dos hijos se distinguen entre sí y se conocen como hijo izquierdo e hijo derecho.

Los árboles binarios de búsqueda son árboles binarios cuyos nodos guardan elementos sobre los cuales hay definido un orden total estricto y que satisfacen la propiedad: el elemento en cada nodo es mayor que todos sus descendientes izquierdos y menor que todos sus descendientes derechos. Equivalentemente, o bien el árbol es vacío, o bien el elemento de la raíz es mayor que los elementos del hijo izquierdo y menor que los elementos del hijo derecho, y recursivamente los dos hijos son a su vez árboles binarios de búsqueda.

Al ser árboles binarios se ofrecen al usuario las operaciones propias de estos:

- Crear el árbol vacío
- Construir un árbol a partir de un elemento y dos árboles (Plantar)
- Consultar la raíz
- Calcular el hijo izquierdo
- Calcular el hijo derecho
- Calcular la altura (número de nodos de la rama más larga)
- Determinar si un árbol es vacío
- Realizar el recorrido en preorden del árbol
- Realizar el recorrido en inorden del árbol
- Realizar el recorrido en postorden del árbol

Las operaciones propias de los arboles binarios de búsqueda que se ofrecen al usuario son:

- Insertar un elemento
- Eliminar un elemento

- Determinar si un elemento pertenece al árbol (Buscar)
- Consultar el menor elemento (Mínimo)
- Consultar el mayor elemento (Máximo)

Al iniciar la estructura de datos solo están activos los botones de las operaciones crear y plantar ya que son las encargadas de construir el árbol.

Sobre el panel de animación podrá verse de forma animada el comportamiento de aplicar estas operaciones. A continuación, se analiza cada una de ellas en detalle:

- **Crear**: al seleccionar esta operación se creará un árbol binario de búsqueda vacío sobre el que se podrá aplicar el resto de las operaciones. Previamente, se deberá indicar el tipo de los elementos que contendrá el árbol binario de búsqueda. Para ello, aparecerá una pequeña ventana, como la de la **figura 66**, donde se elegirá el tipo de los datos.

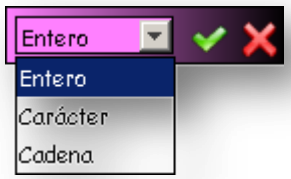


Figura 66 – Ventana donde el usuario elige el tipo de la estructura.

Una vez que se ha ejecutado la operación, se deshabilitará su botón y el botón de plantar, ya que no se podrá crear por segunda vez.

- **Plantar**: al seleccionar esta operación se creará un árbol binario de búsqueda, a partir de un elemento (raíz) y de dos árboles binarios de búsqueda, sobre el que se podrá aplicar el resto de las operaciones. Previamente, se deberá indicar:
  - El tipo de los elementos que contendrá el árbol binario de búsqueda. Para ello, aparecerá una pequeña ventana, como la de la **figura 66**, donde se elegirá el tipo de los datos.
  - El elemento que ocupará la raíz y sus hijos. Para ello, aparecerá una nueva ventana donde se podrá introducir el dato que ocupará la raíz y los nombres de los árboles que serán sus hijos.

Para la raíz, solo se permite introducir los caracteres que formen datos del tipo indicado en la ventana anterior con un máximo de 3 caracteres.

Esta operación, solamente, tiene sentido cuando el elemento a colocar en la raíz es mayor que todos los elementos del hijo izquierdo y menor que los del hijo derecho. Para que no se produzcan situaciones de error, según se introduce el dato de la raíz se actualizan los desplegables, donde se eligen los hijos, con los árboles binarios de búsqueda que cumplan:

- Son árboles ya creados en **VEDYA**.
- Son del tipo indicado anteriormente.
- Su menor elemento es mayor que el elemento de la raíz o su mayor elemento es menor que la raíz.

La ventana donde se introducen estos datos se muestra en la ***figura 67***.

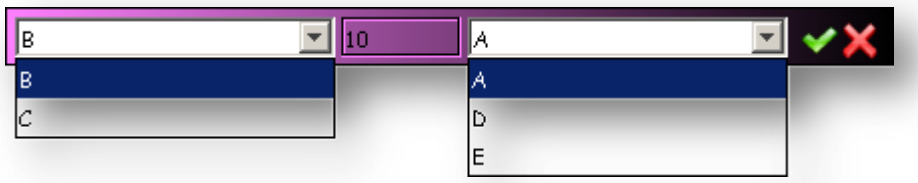


Figura 67 – Ventana donde el usuario elige la raíz y los árboles que quiere plantar.

Al realizar la operación plantar, se efectúa una copia de los árboles que van a ser sus hijos. De esta forma se garantiza la independencia entre los árboles originales y el nuevo árbol creado.

A esta operación se le asocia dos botones situados en la barra de estados de la pestaña, que si pulsamos sobre ellos nos muestran el hijo izquierdo y el hijo derecho en el momento en que se plantó el árbol. No se permite que se modifiquen estos árboles, por lo que se han inhabilitado todas las opciones sobre ellos.

Una vez que se ha ejecutado la operación, se deshabilitará su botón y el botón de crear, ya que no se podrá crear un árbol binario de búsqueda por segunda vez.

- **Raíz?**: al seleccionar esta operación se señalará el elemento situado en la raíz del árbol. Para que ésta operación esté disponible será necesario que el árbol binario de búsqueda se haya creado y que contenga algún elemento.

- **Hijo izquierdo?**: *al seleccionar esta operación se señalarán los elementos que pertenecen al hijo izquierdo del árbol. Para que ésta operación esté disponible será necesario que el árbol binario de búsqueda se haya creado, mediante la operación crear o plantar, y que contenga algún elemento.*
- **Hijo derecho?**: *al seleccionar esta operación se señalarán los elementos que pertenecen al hijo derecho del árbol. Para que ésta operación esté disponible será necesario que el árbol binario de búsqueda se haya creado, mediante la operación crear o plantar, y que contenga algún elemento.*
- **Altura**: *si se selecciona esta operación, la aplicación indicará la altura del árbol binario de búsqueda. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol mediante la operación crear o plantar.*
- **Es vacío?**: *si se selecciona esta operación, la aplicación indicará si el árbol binario de búsqueda está vacío o si, por el contrario, contiene algún elemento. El árbol estará vacío nada mas crearlo o cuando se hayan eliminado todos sus elementos. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol mediante la operación crear o plantar.*
- **Preorden**: *si se selecciona esta operación se mostrará el recorrido en preorden del árbol, siguiendo la regla de primero recorrer la raíz, luego el hijo izquierdo y por último el hijo derecho. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol mediante la operación crear o plantar.*
- **Inorden**: *si se selecciona esta operación se mostrará el recorrido en inorden del árbol, siguiendo la regla de primero recorrer el hijo izquierdo, luego la raíz y por último el hijo derecho. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol mediante la operación crear o plantar.*
- **Postorden**: *si se selecciona esta operación se mostrará el recorrido en inorden del árbol, siguiendo la regla de primero recorrer el hijo izquierdo, luego el hijo derecho y por último la raíz. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol mediante la operación crear o plantar.*
- **Insertar**: *para que esta operación pueda estar disponible será necesario que el árbol binario de búsqueda se haya creado con anterioridad mediante la*

operación crear o plantar. Al seleccionar esta operación podrá verse de manera animada cómo se introduce un nuevo elemento en el árbol, manteniendo la relación de orden, siempre y cuando el elemento no estuviera ya contenido en el árbol.

Es el usuario el que indica el dato que se introduce, siendo éste del tipo que se indicó al crear el árbol binario de búsqueda. Para ello, aparecerá una ventana, como la que se muestra en la **figura 68**, donde se podrá introducir el dato. Solo se permite introducir los caracteres que formen datos del tipo indicado en la operación crear, por ejemplo, si al crear se indico que los datos fuesen enteros solo se permiten introducir los caracteres del 0 al 9 y "–". Además, por razones de espacio en el elemento gráfico, solo se permiten, como máximo, elementos de longitud 3.

Figura 68 – Ventana donde el usuario introduce los datos de entrada de la estructura de datos.

No se han puesto limitaciones al número de elementos que puede contener un árbol binario de búsqueda, por lo que este botón no se volverá a deshabilitar.

- **Eliminar**: al seleccionar esta operación podrá verse de manera animada cómo se busca el elemento a eliminar. Si el elemento no está no se hará nada. Sin embargo, si el elemento que queremos eliminar está en el árbol, y teniendo en cuenta que se debe mantener el orden en el árbol, se hará:
  - Si el elemento es una hoja, simplemente se elimina
  - Si el elemento es un nodo con hijo izquierdo o hijo derecho, el subárbol del nodo sustituye al propio nodo.
  - Si el elemento es un nodo con hijo izquierdo e hijo derecho, se busca el mínimo de su hijo derecho. Una vez encontrado se intercambia su valor con el del nodo que queremos eliminar, y se elimina el nodo como en uno de los dos casos anteriores.

  Es el usuario el que indica el elemento a eliminar. Para ello, aparecerá una ventana, como la que se muestra en la **figura 68.**

Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol binario de búsqueda mediante la operación crear o plantar.

- **Está?:** al seleccionar esta operación podrá verse de manera animada cómo se busca el elemento introducido por el usuario mediante una ventana como la que se muestra en la **figura 68**. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol binario de búsqueda mediante la operación crear o plantar.
- **Mínimo?**: al seleccionar esta operación podrá verse de manera animada cómo se busca el elemento mínimo y una vez encontrado se señalará. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol binario de búsqueda, mediante la operación crear o plantar, y que contenga algún elemento.
- **Máximo?**: al seleccionar esta operación podrá verse de manera animada cómo se busca el elemento máximo y una vez encontrado se señalará. Para que el botón de esta operación esté habilitado es necesario que se haya creado el árbol binario de búsqueda, mediante la operación crear o plantar, y que contenga algún elemento.

A continuación se muestra como se representa gráficamente los árboles binarios de búsqueda para la visualización modo usuario y las animaciones que se realizan.

3.3.1.1.- Visualización "Modo Usuario"

En esta visualización queremos mostrar la filosofía de los árboles binarios de búsqueda, es decir, que los elementos están ordenados de forma que el valor de un nodo siempre va a ser mayor que todos los nodos de su hijo izquierdo y menor que todos los nodos de su hijo derecho. Por ello, en las operaciones características de los árboles binarios de búsqueda, hemos querido animar todas las búsquedas que se realizan para ejecutar las dichas operaciones.

- **Crear**: *al seleccionar esta operación se creará un árbol binario de búsqueda vacío, y se mostrarán los controles de tamaño y posición que, a la postre, nos permitirán modificar el aspecto del árbol, (ver* ***figura 69****)*

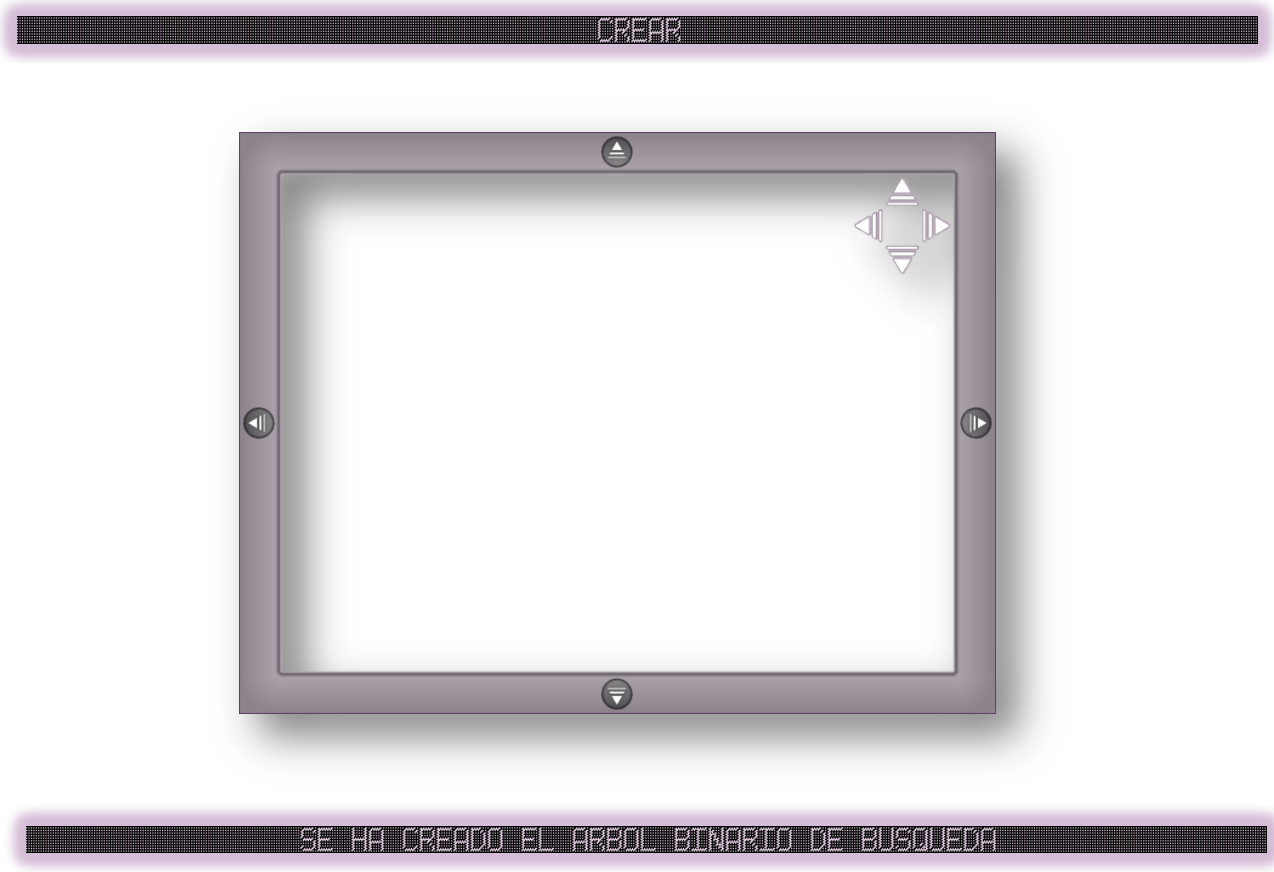


Figura 69 – *Animación de la operación Crear. ABB. Visualización de Usuario.*

- **Plantar**: *esta operación se comentará después explicar el funcionamiento del el resto de las operaciones.*

- **Insertar**: *al seleccionar esta operación se verá cómo se inserta el elemento seleccionado, de forma que él mismo busca su posición en el árbol. En las* ***figuras 70****,* ***71****,* ***72*** *y* ***73*** *se muestran diferentes secuencias de inserción de elementos.*

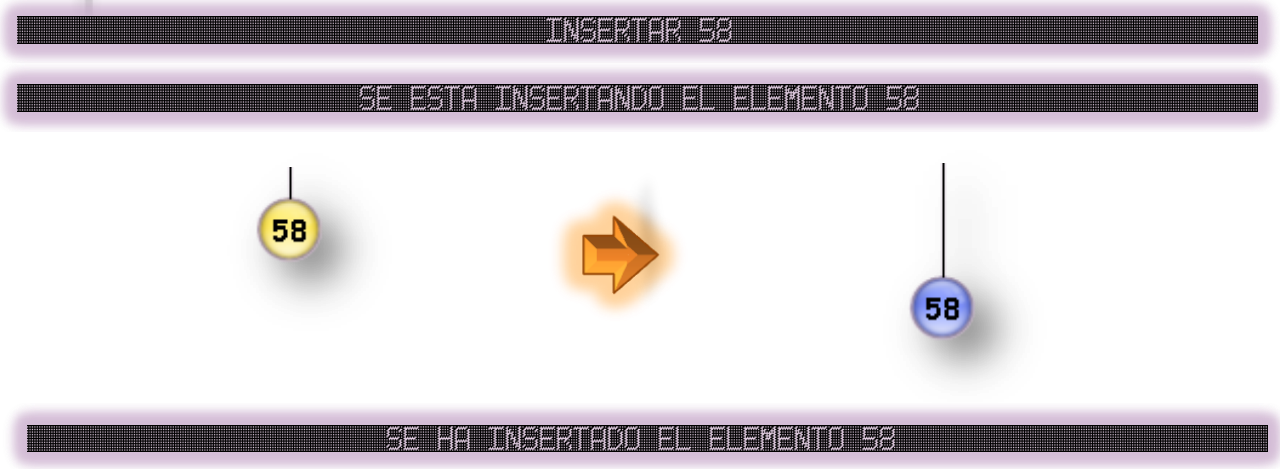


Figura 70 – *Animación de la operación Insertar el elemento 58. ABB. Visualización de Usuario.*

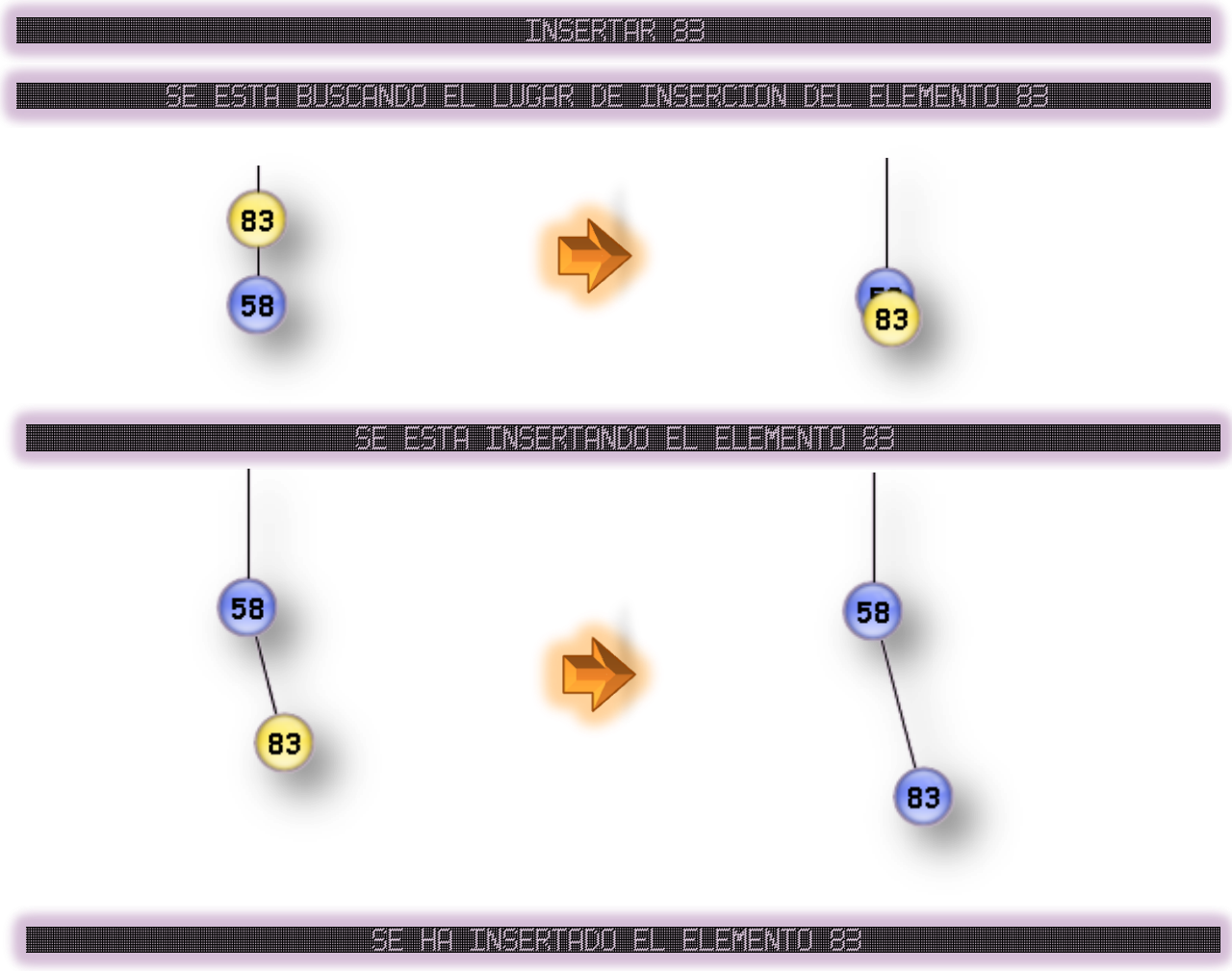


Figura 71 – *Animación de la operación Insertar el elemento 83. ABB. Visualización de Usuario.*

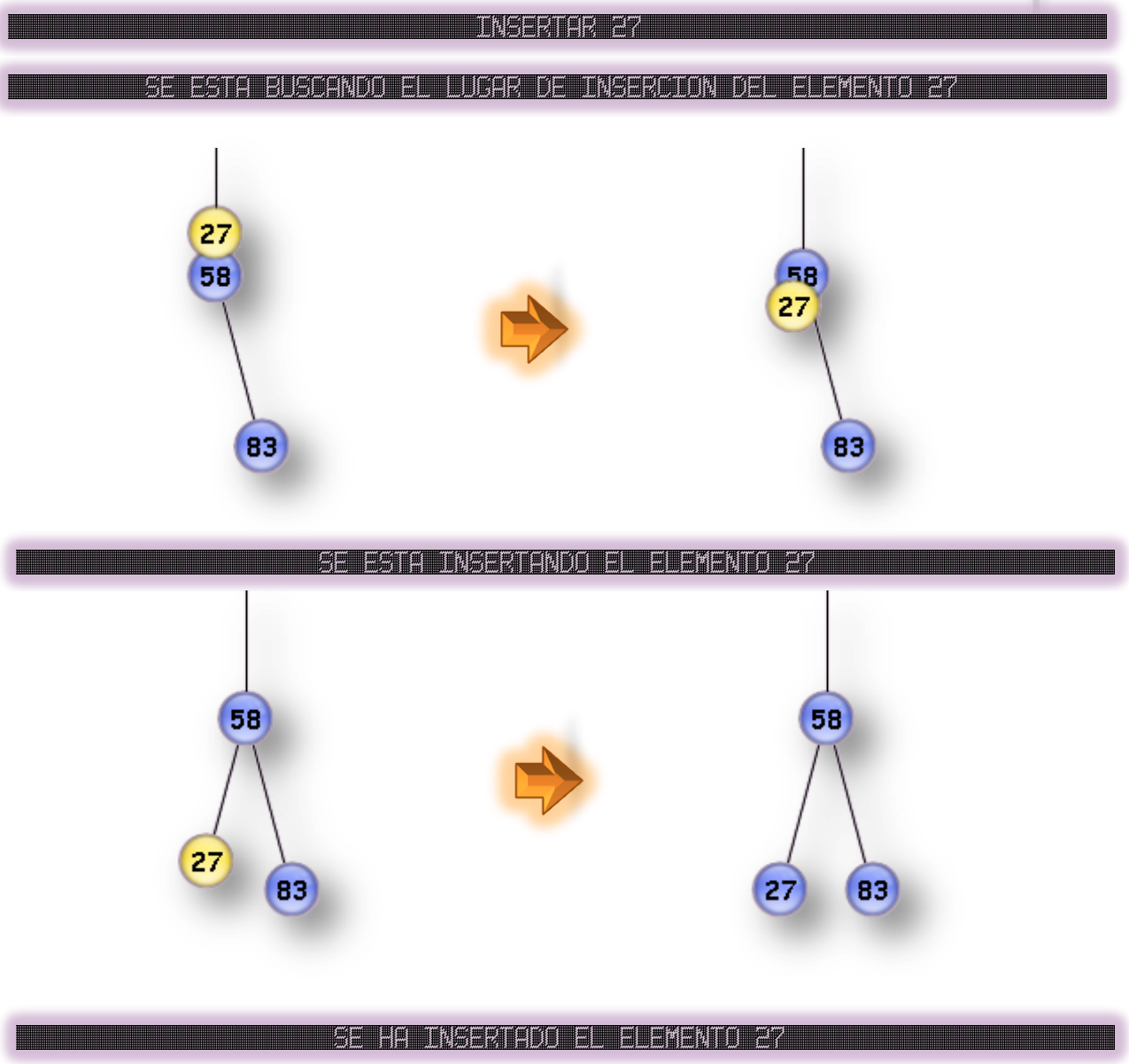


Figura 72 – *Animación de la operación Insertar el elemento 27. ABB. Visualización de Usuario.*

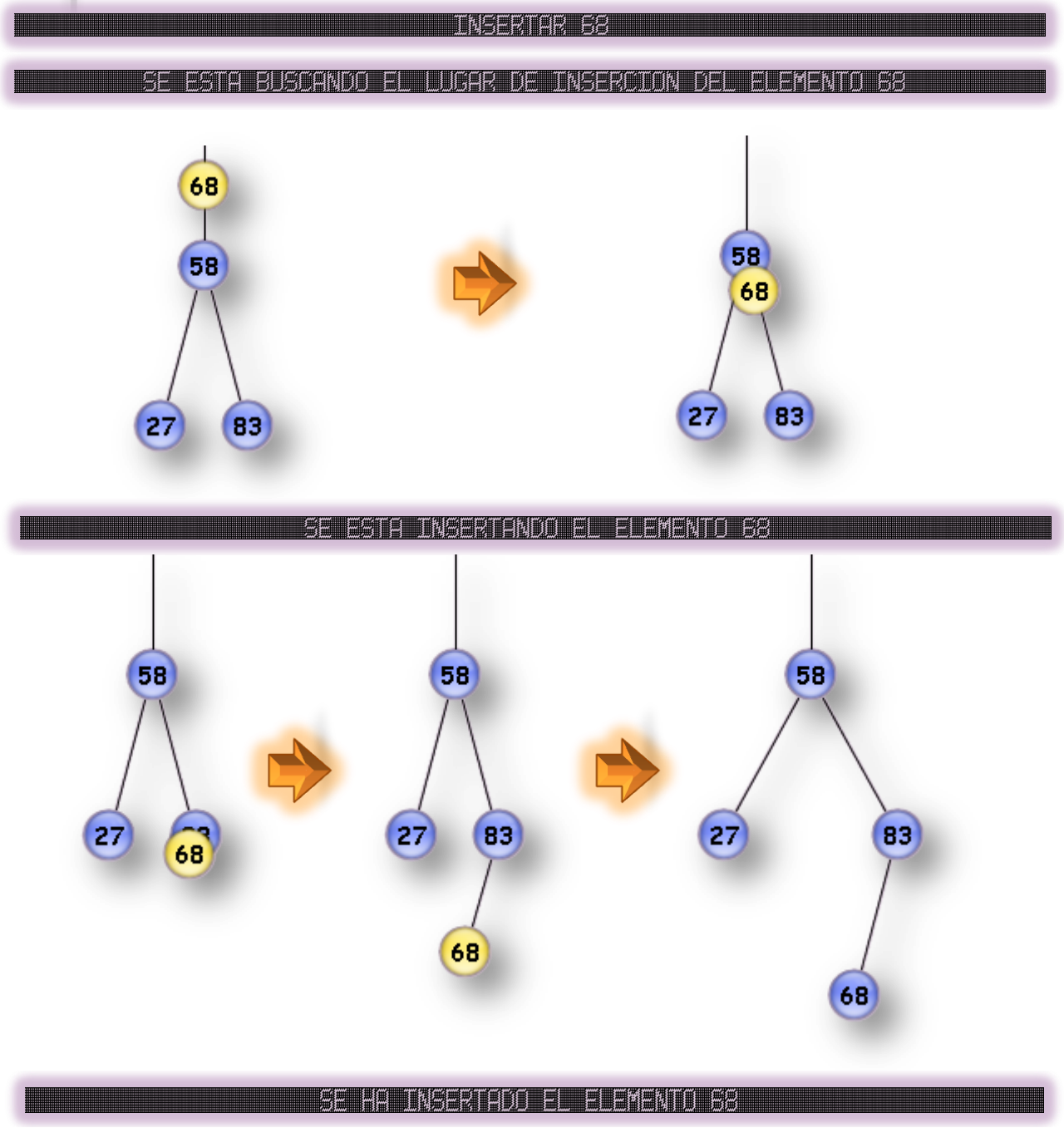


Figura 73 – *Animación de la operación Insertar el elemento 68. ABB. Visualización de Usuario.*

Para continuar la explicación del resto de operaciones del árbol binario de búsqueda, se han insertado los siguientes elementos en orden:

15, 9, 60, 90, 86, 18, 80, 74, 82

Después de insertarlos, el árbol binario de búsqueda queda como se muestra en la **figura 74**.

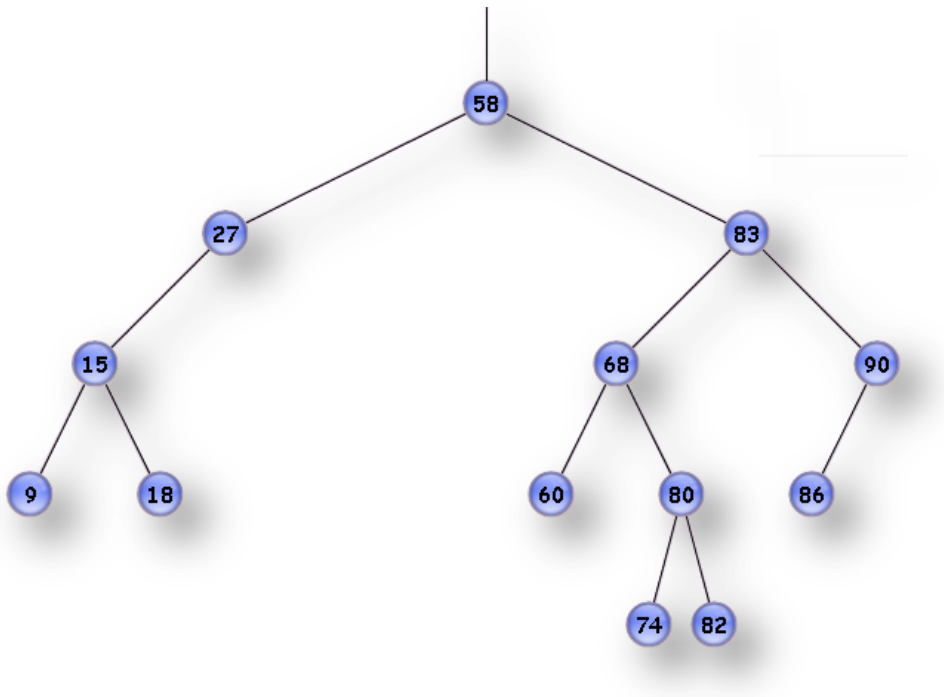
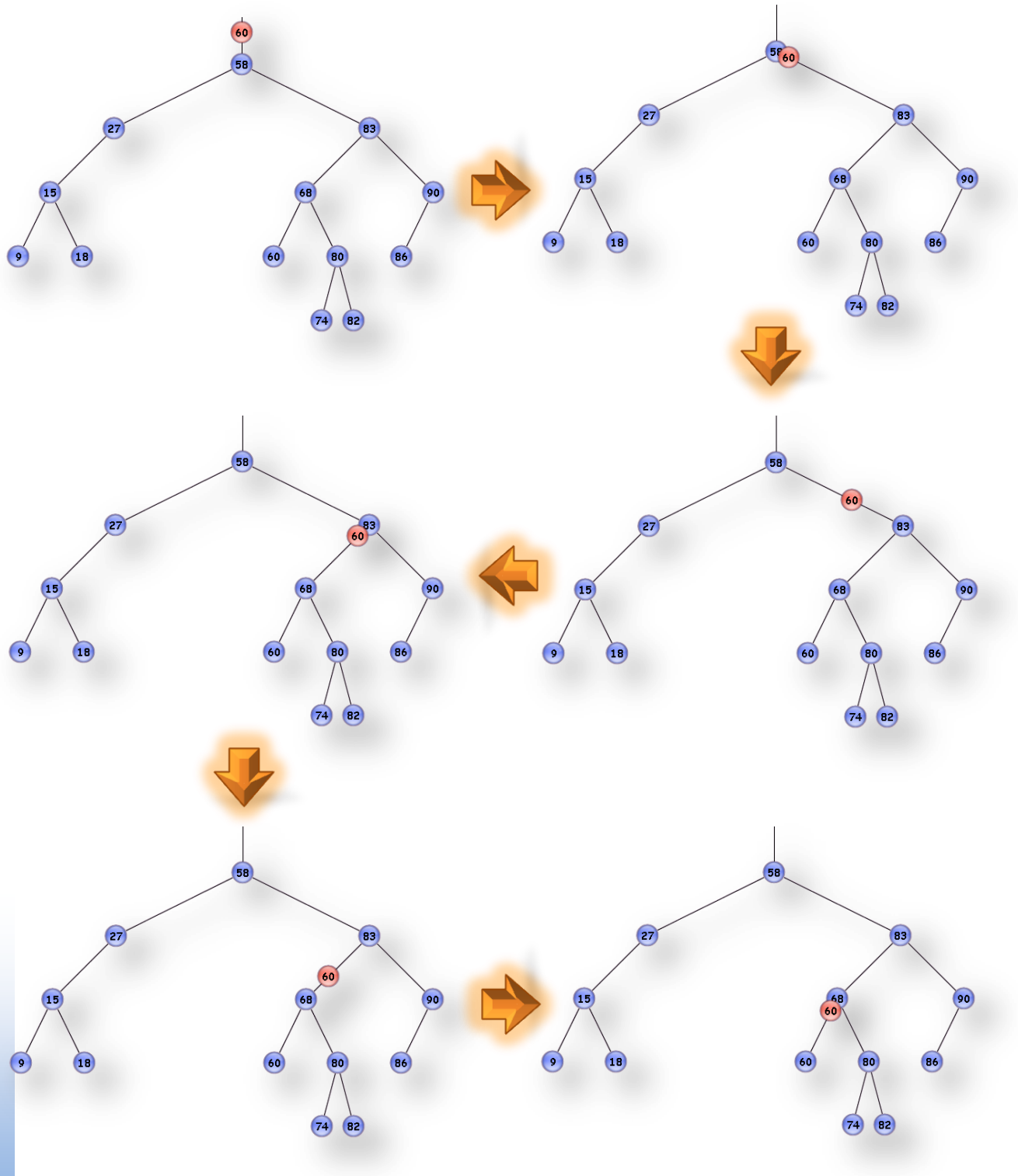


Figura 74 – Insertamos varios elementos. ABB. Visualización de Usuario.

- **Eliminar**: al seleccionar esta operación podrá verse de manera animada la eliminación del elemento, de forma que se mantenga el orden del árbol. En las **figuras 75**, **76** y **77** se muestran las distintas secuencias que pueden producirse al eliminar un elemento.

ELIMINAR 60

SE ESTA BUSCANDO EL ELEMENTO 60 PARA SU ELIMINACION



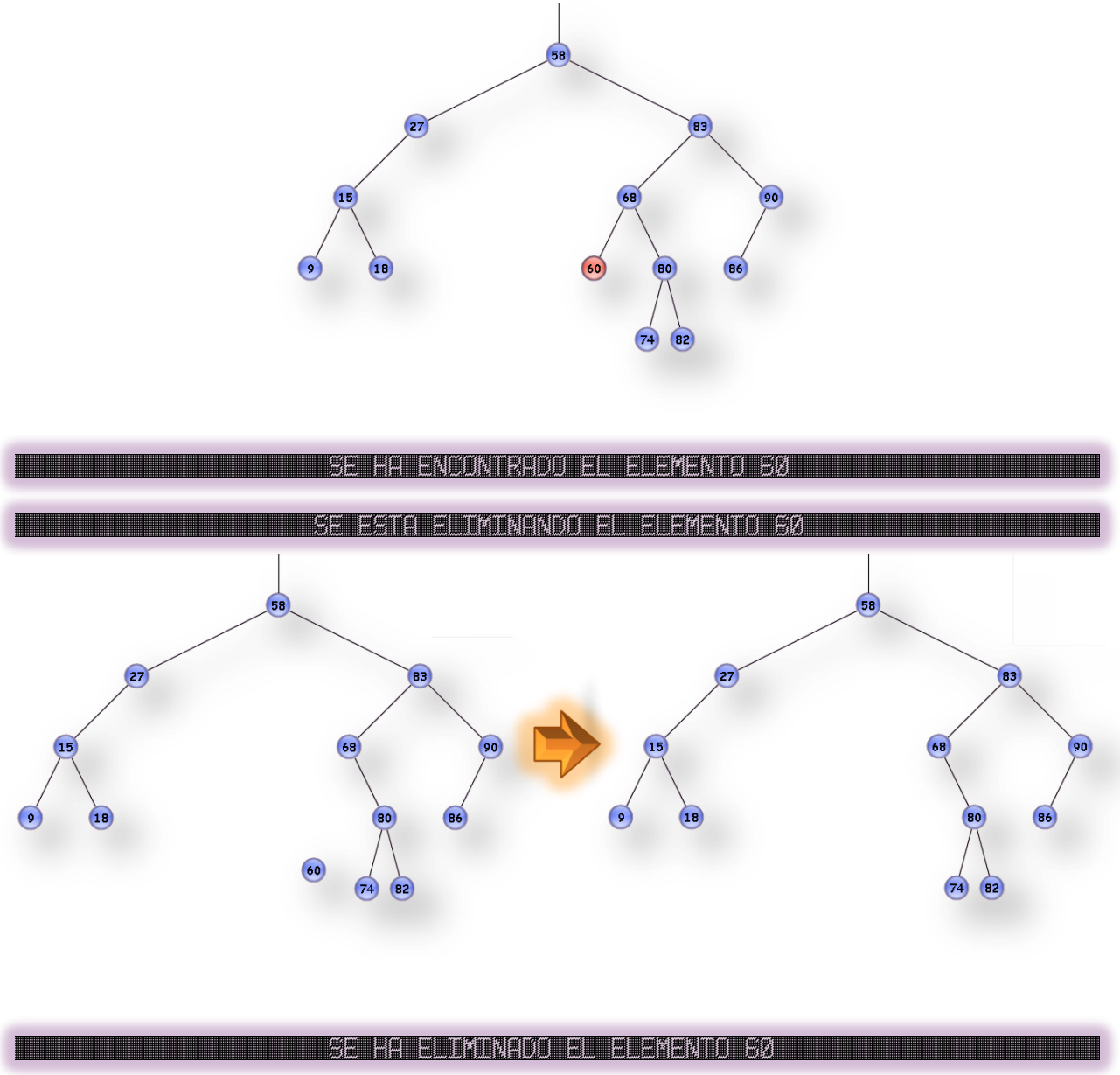
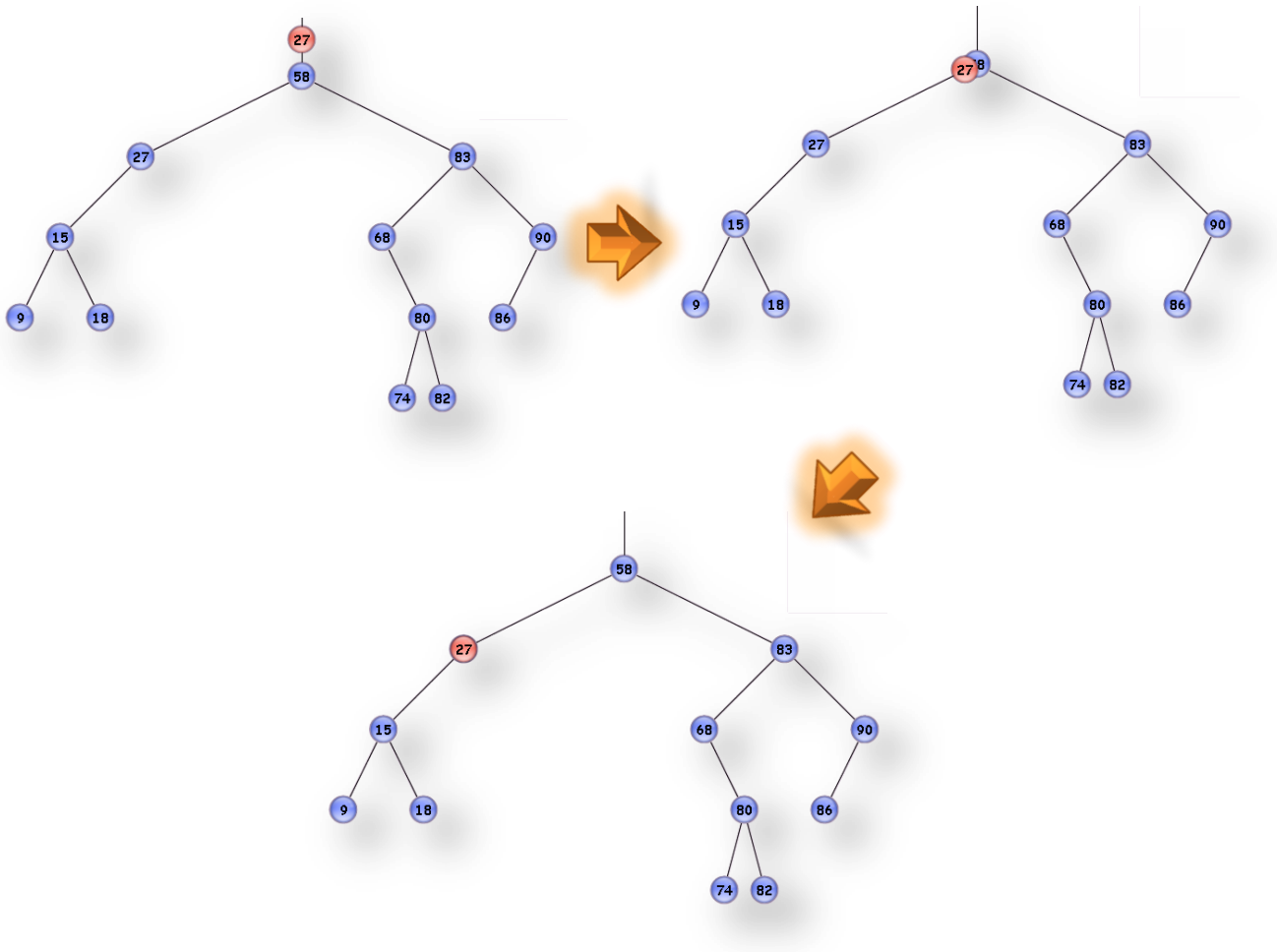


Figura 75 – *Animación de la operación eliminar el elemento 60. ABB. Visualización de Usuario.*

ELIMINAR 27

SE ESTA BUSCANDO EL ELEMENTO 27 PARA SU ELIMINACION



SE HA ENCONTRADO EL ELEMENTO 27

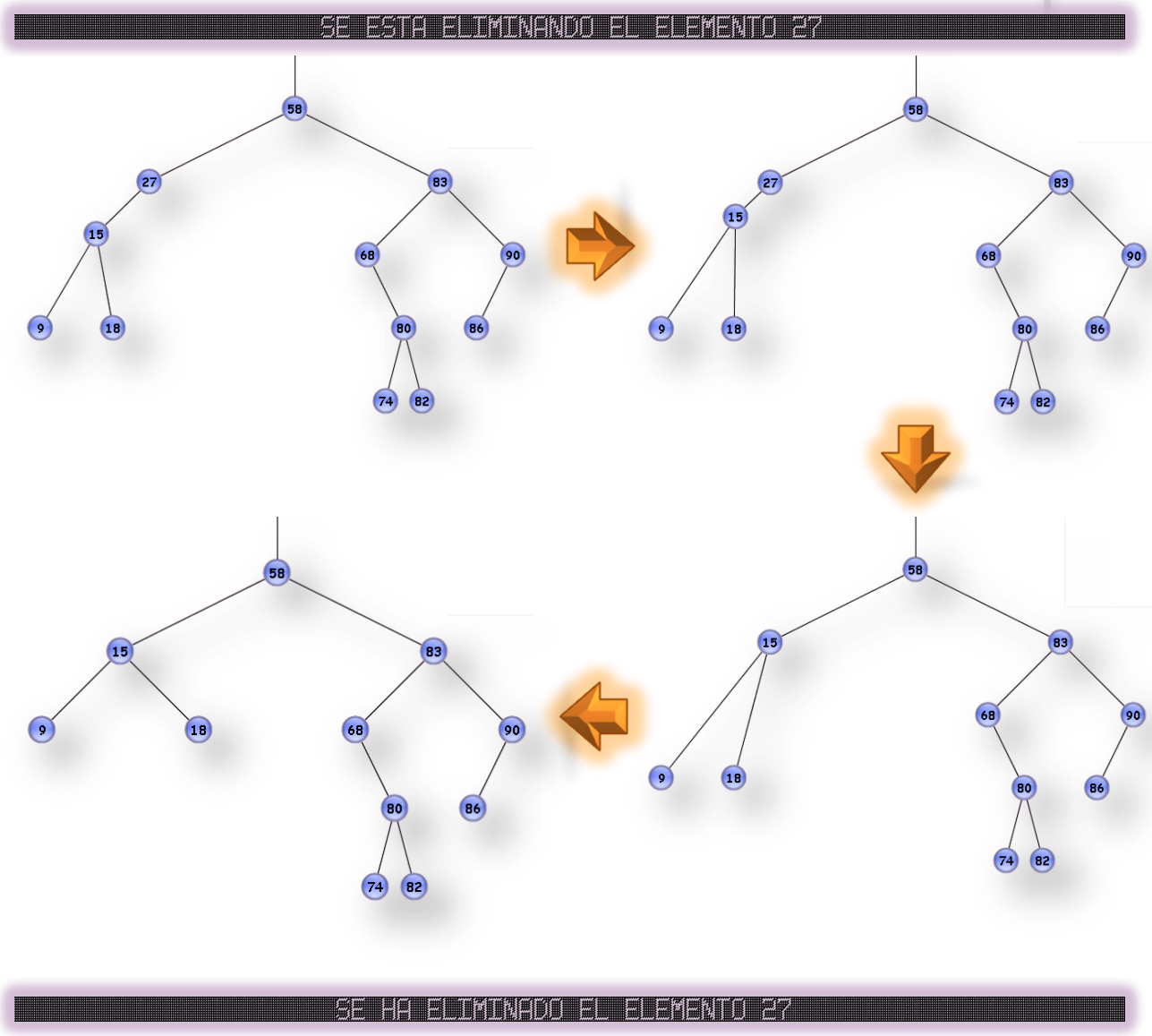
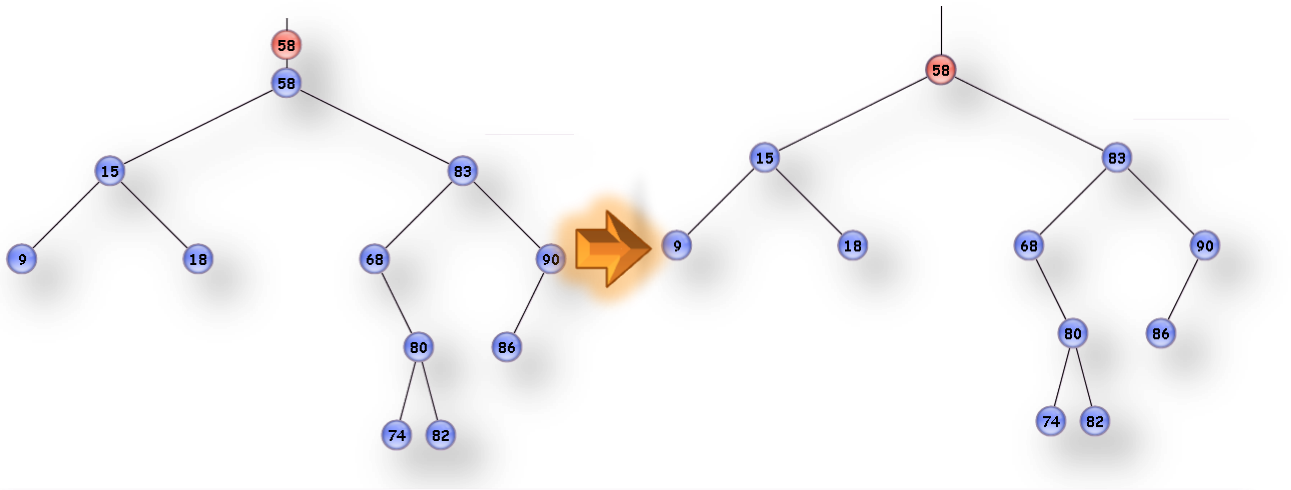


Figura 76 – *Animación de la operación eliminar el elemento 27. ABB. Visualización de Usuario.*

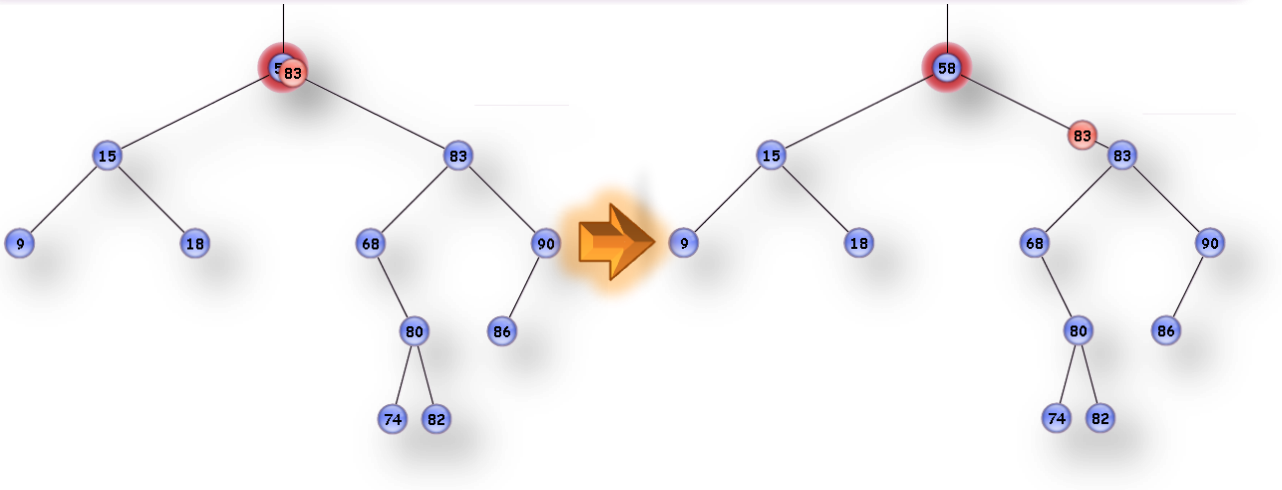
ELIMINAR 58

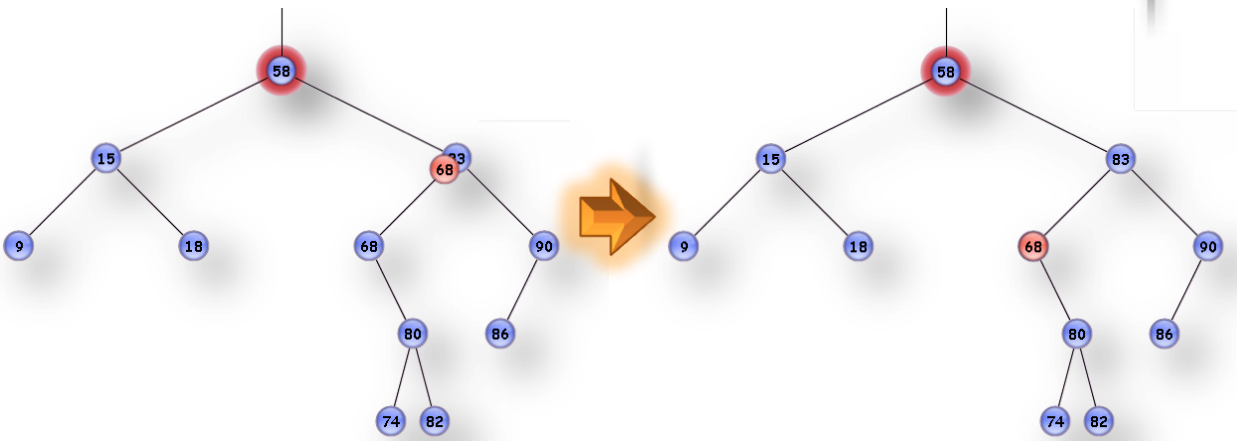
SE ESTA BUSCANDO EL ELEMENTO 58 PARA SU ELIMINACION



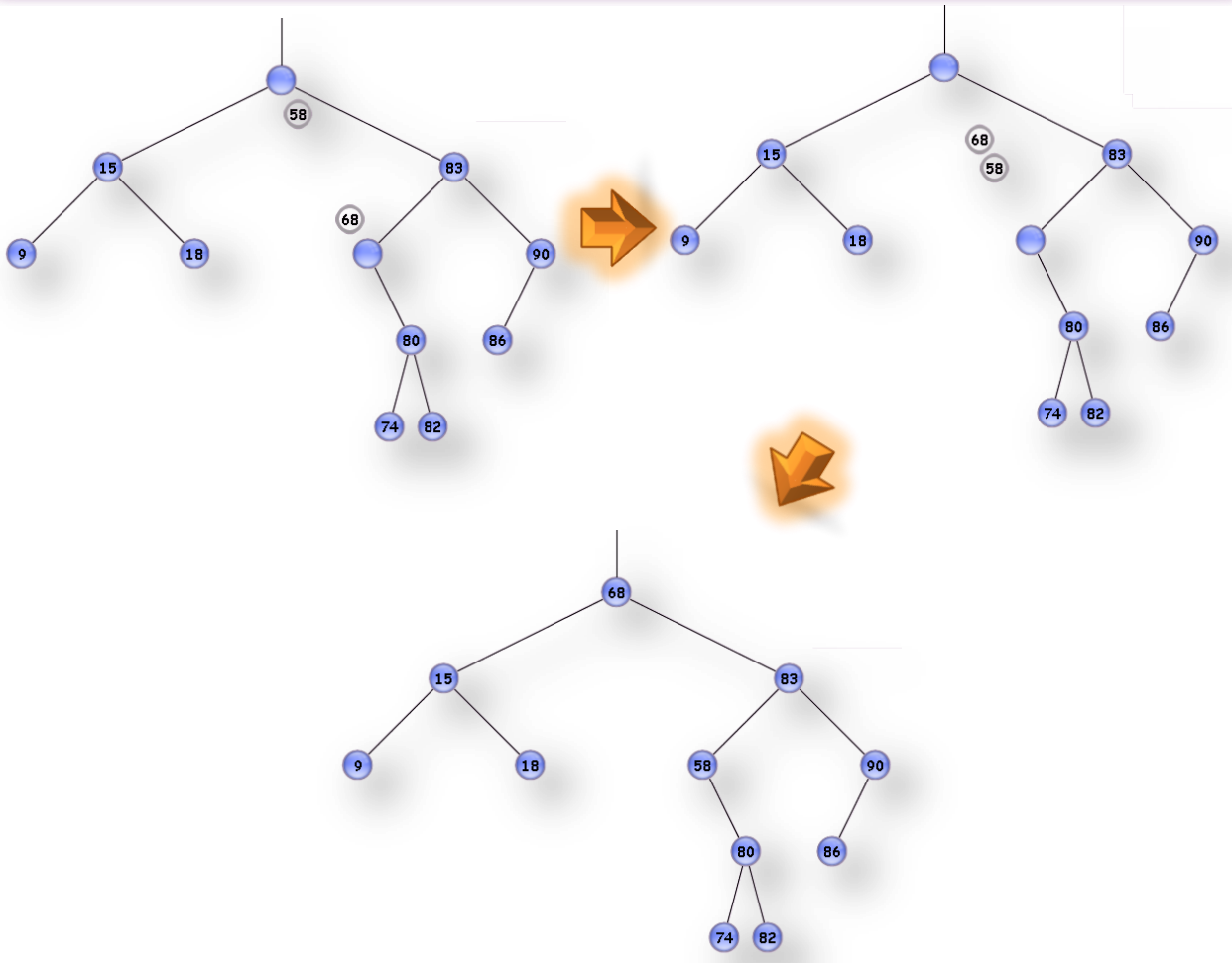
SE HA ENCONTRADO EL ELEMENTO 58

SE ESTA BUSCANDO EL MINIMO DEL HIJO DERECHO DEL ELEMENTO 58





Se intercambia el elemento a eliminar y el mínimo de su hijo derecho



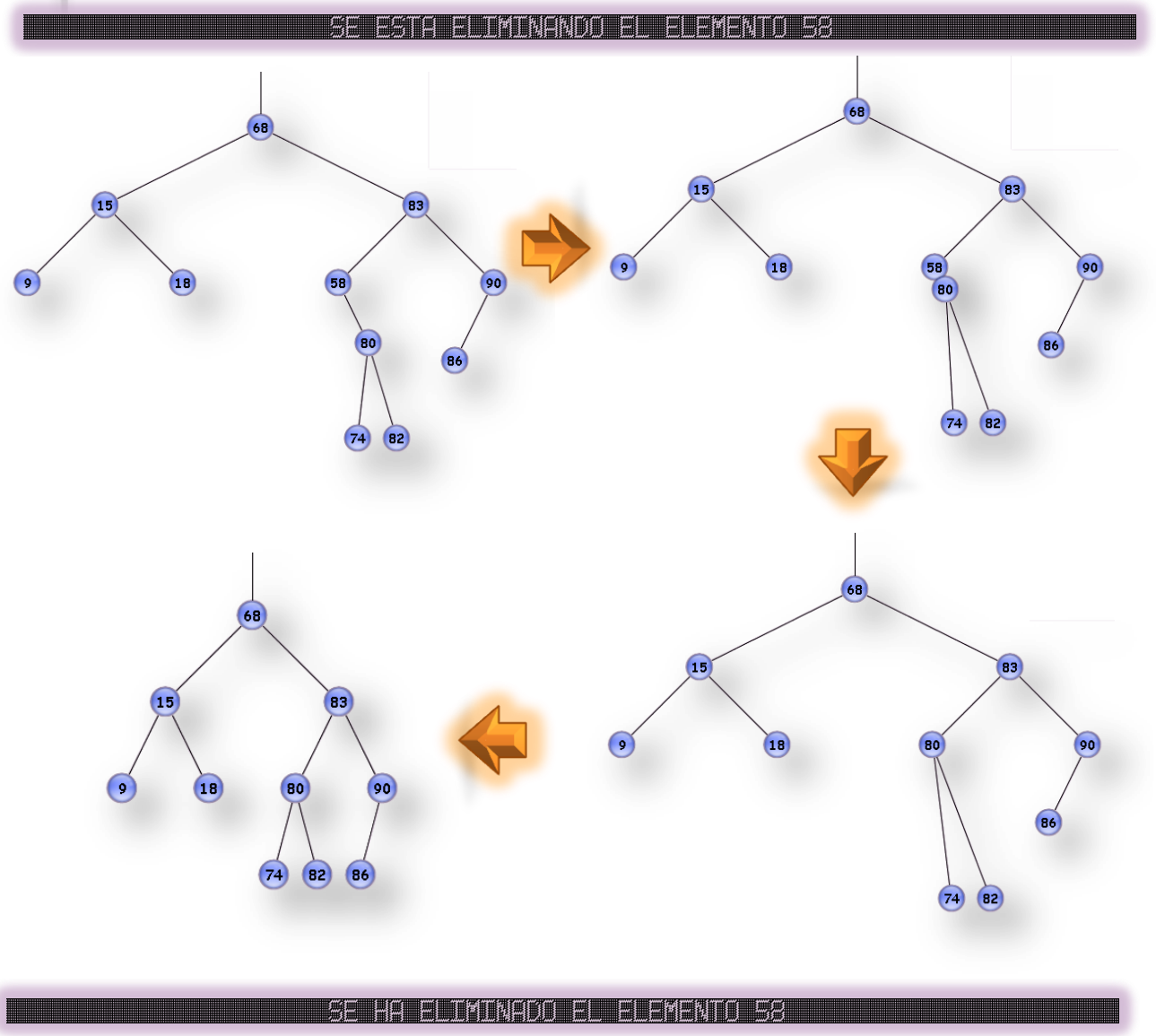


Figura 77 – *Animación de la operación eliminar el elemento 58. ABB. Visualización de Usuario.*

- **Raíz?**: *al seleccionar esta operación se iluminará el elemento situado en la raíz del árbol con una luz roja, como se indica en la* ***figura 78***.

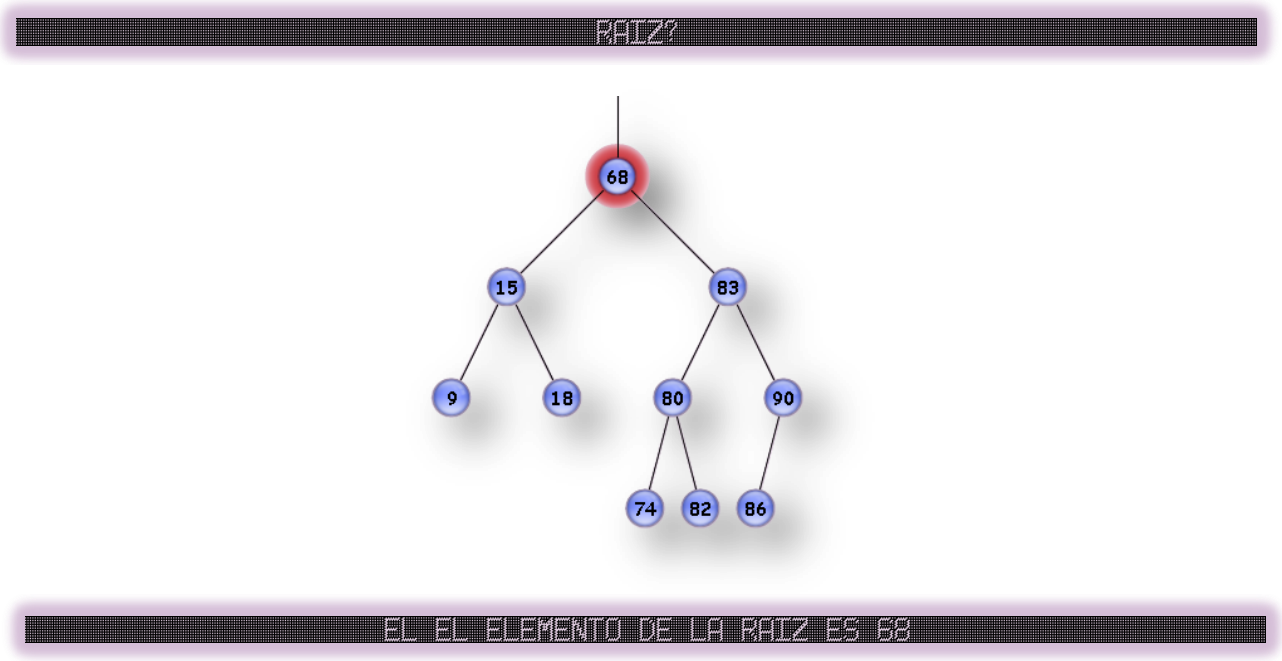


Figura 78 – Animación de la operación Raiz?. ABB. Visualización de Usuario.

- **Hijo izquierdo?**: *al seleccionar esta operación se iluminarán todos los elementos del hijo izquierdo de la raíz del árbol con una luz roja. (Ver* ***figura 79***)

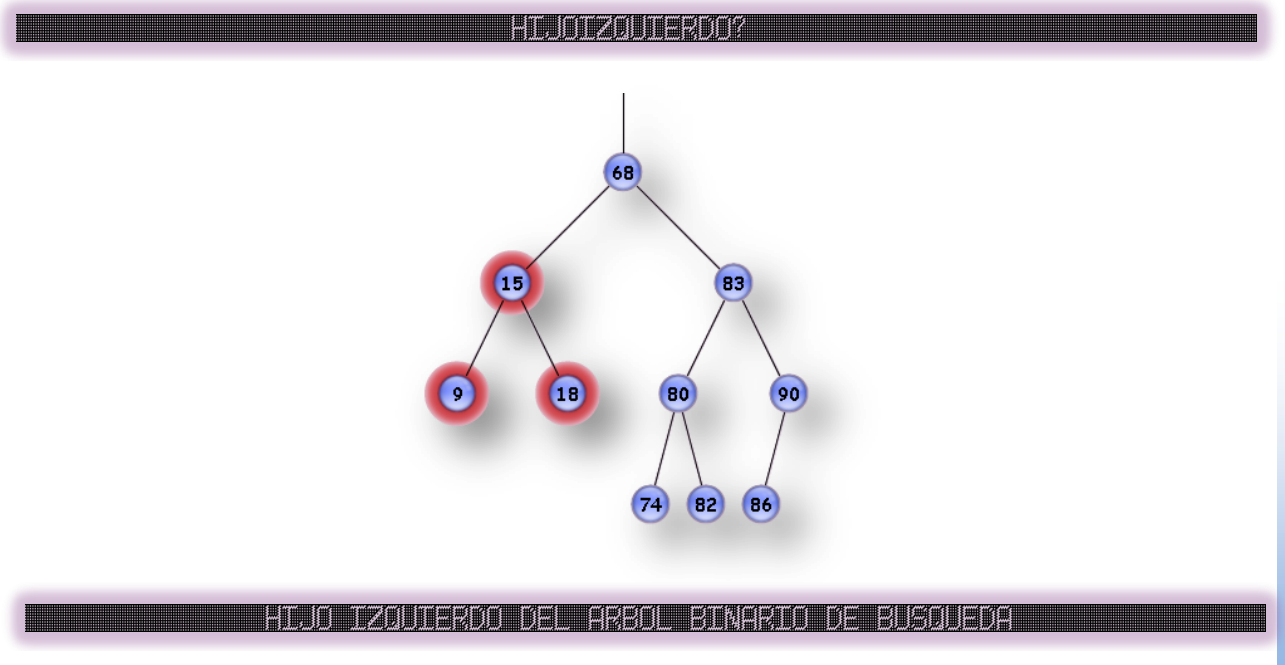


Figura 79 – Animación de la operación HijoIzquierdo?. ABB. Visualización de Usuario.

- **Hijo derecho?**: *al seleccionar esta operación se iluminarán todos los elementos del hijo derecho de la raíz del árbol con una luz roja. (Ver* ***figura 80****).*

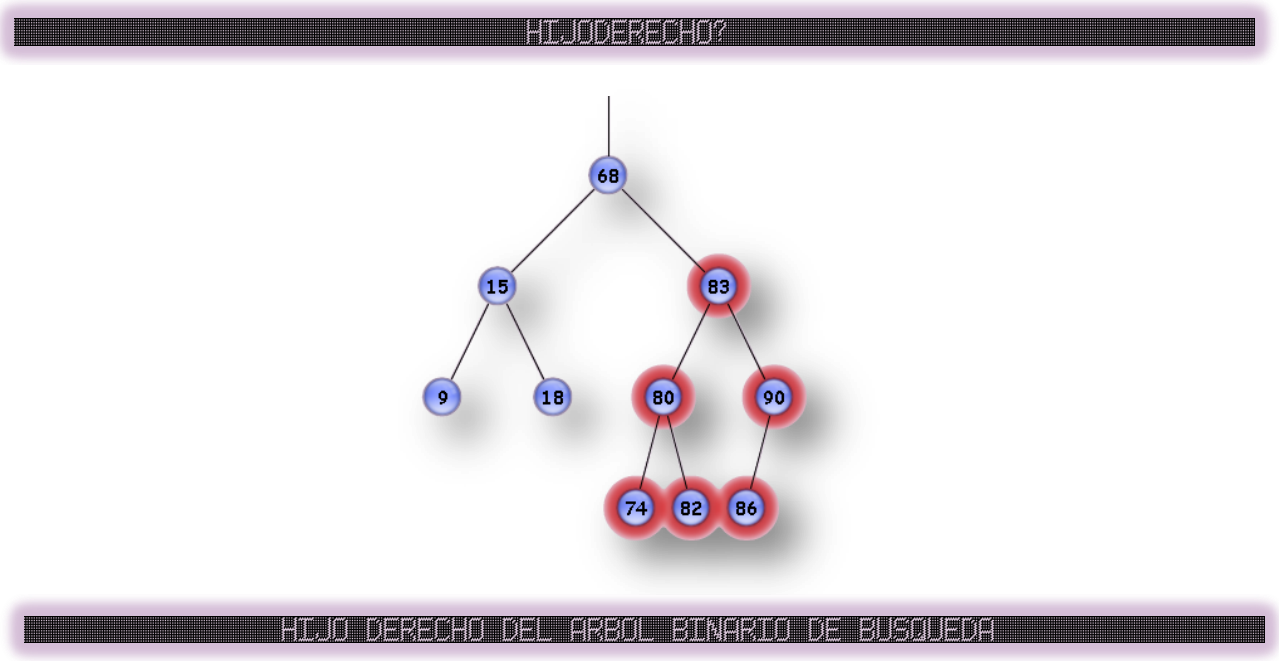


Figura 80 – *Animación de la operación HijoDerecho?. ABB. Visualización de Usuario.*

- **Altura**: *al seleccionar esta operación se indicarán cada uno de los niveles del árbol, tal y como se muestra en la* ***figura 81****.*

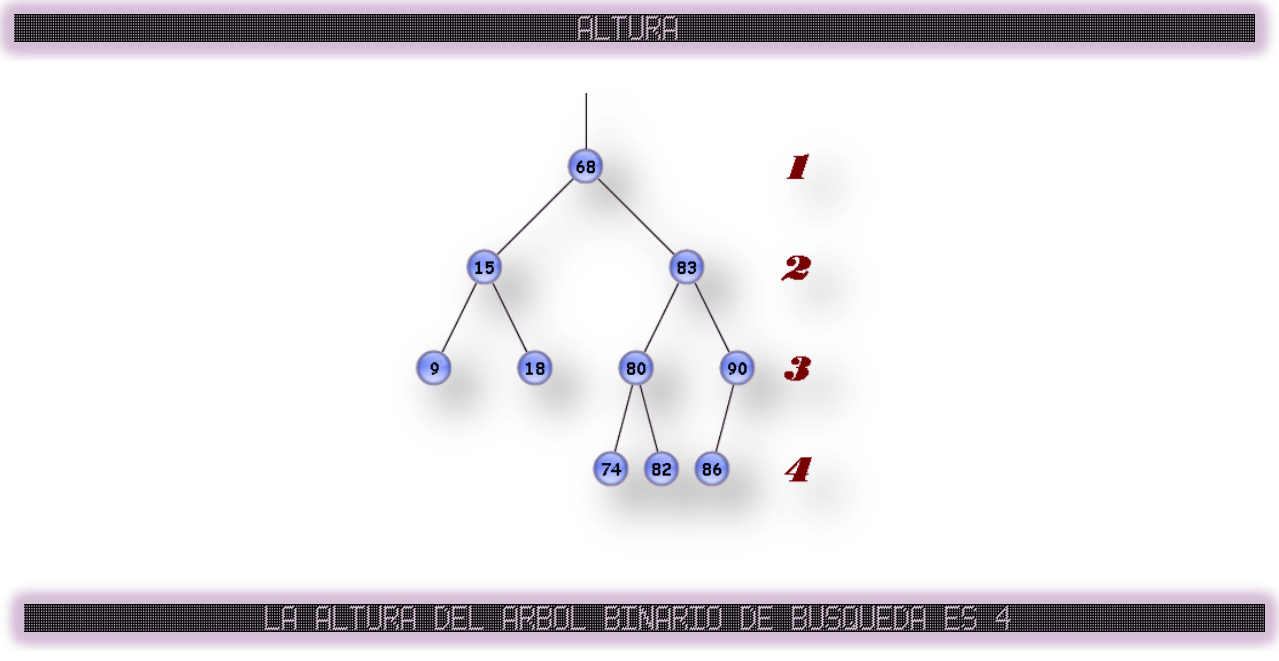


Figura 81 – *Animación de la operación Altura. ABB. Visualización de Usuario.*

- **Es vacío?**: Al seleccionar esta operación se muestra en el cuadro de dialogo si el árbol está o no vacío. (Ver **figura 82**).

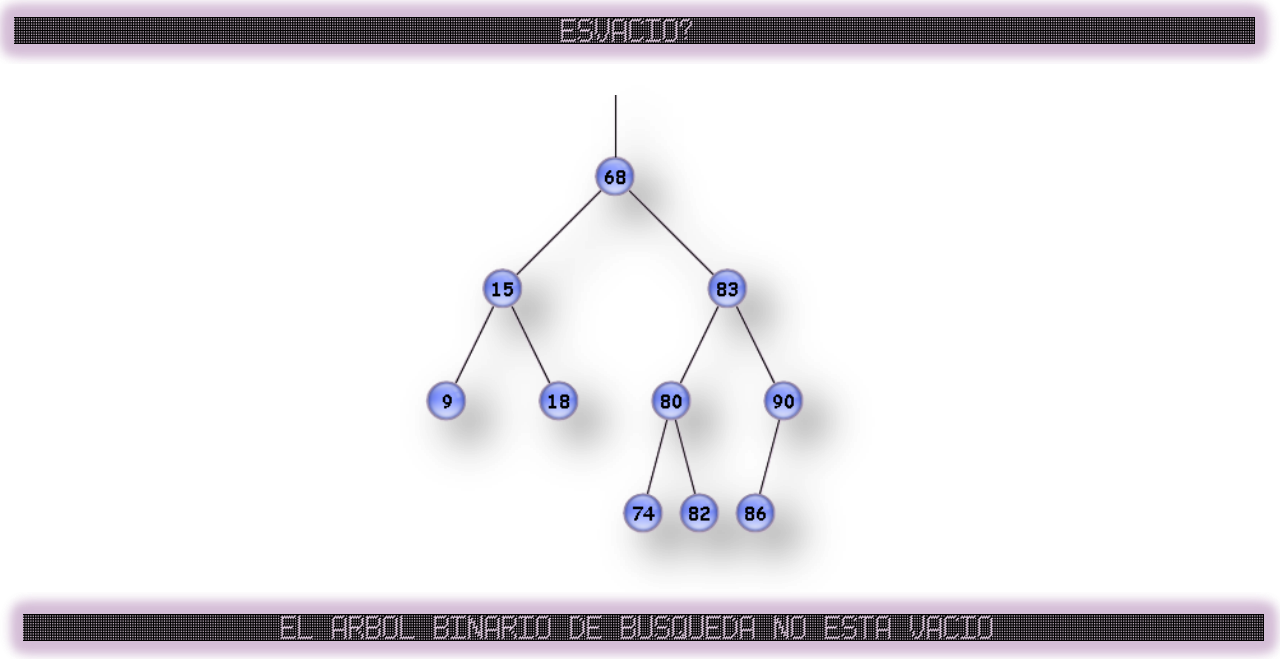
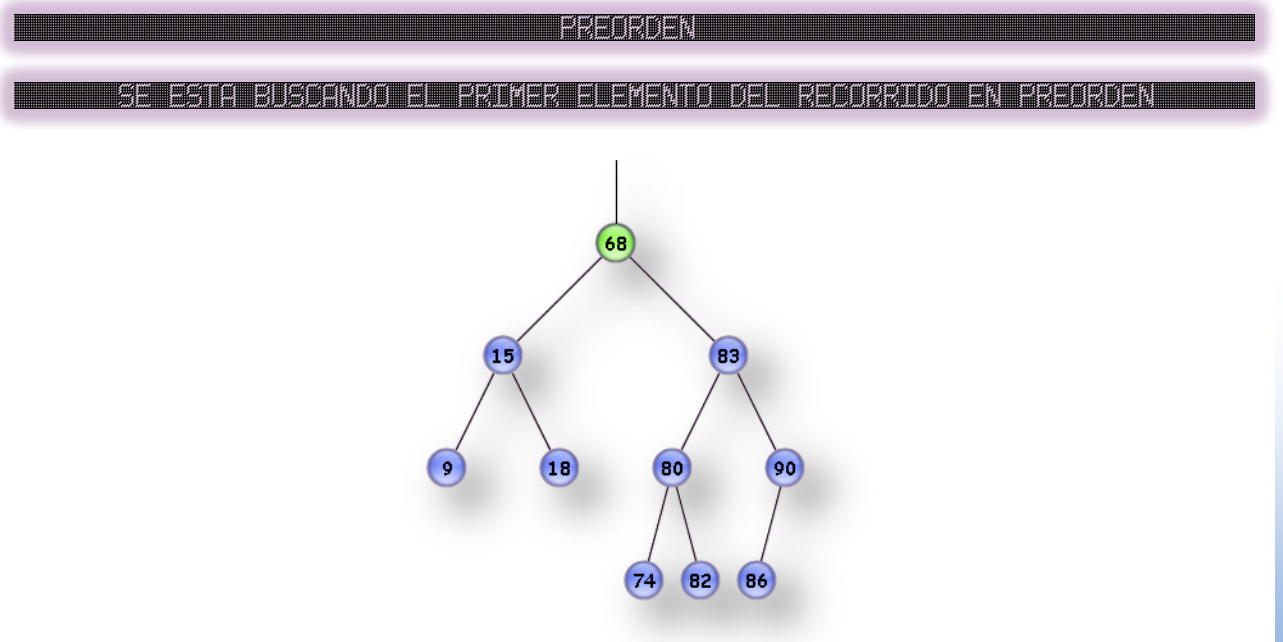
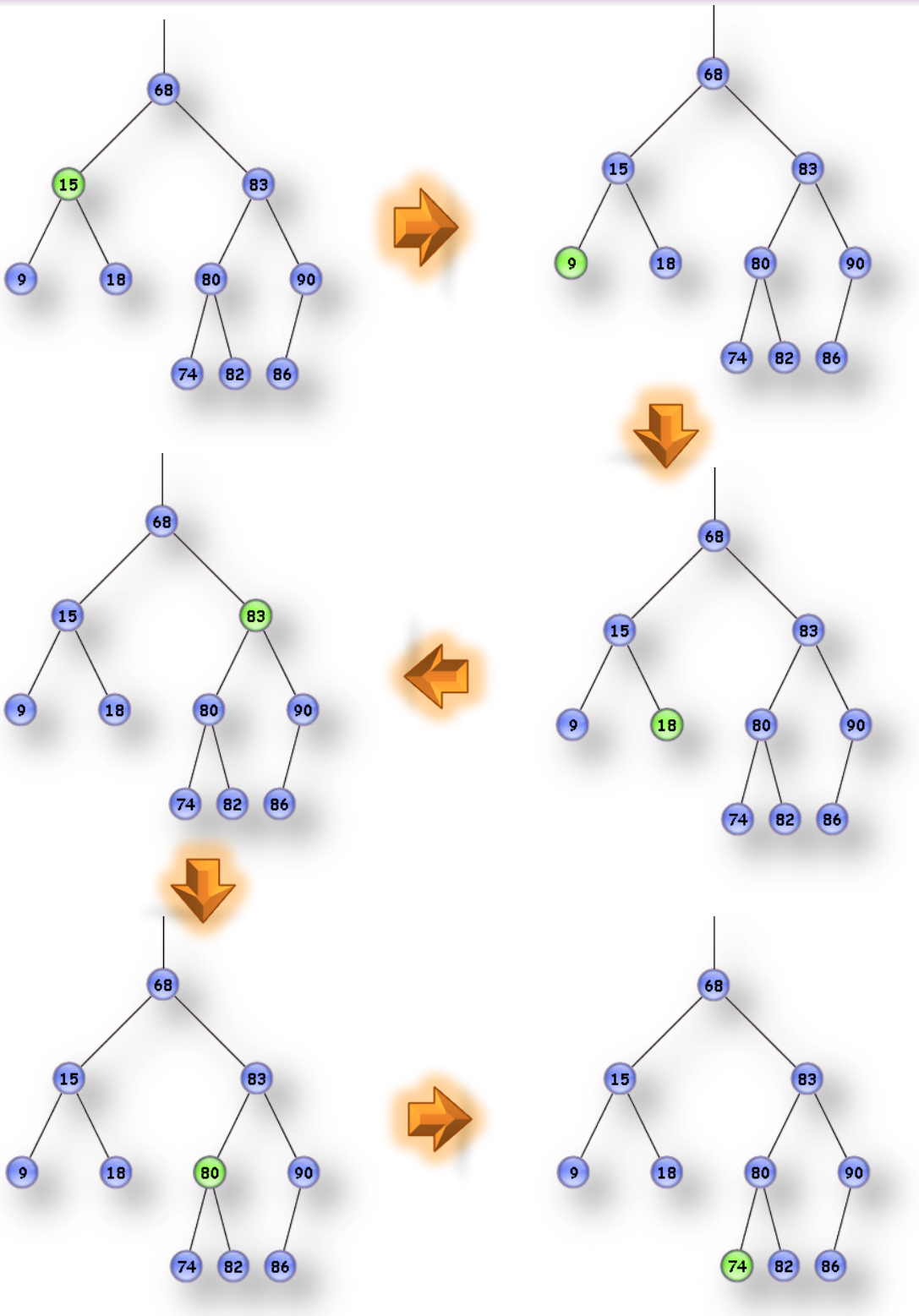


Figura 82 – Animación de la operación EsVacio?. ABB. Visualización de Usuario.

- **Preorden**: al seleccionar esta operación se mostrará, con ayuda de un elemento auxiliar, el recorrido en preorden del árbol. El elemento auxiliar se desplazará de nodo en nodo en un recorrido en preorden. (Ver **figura 83**)



Se está buscando el siguiente elemento del recorrido en preorden



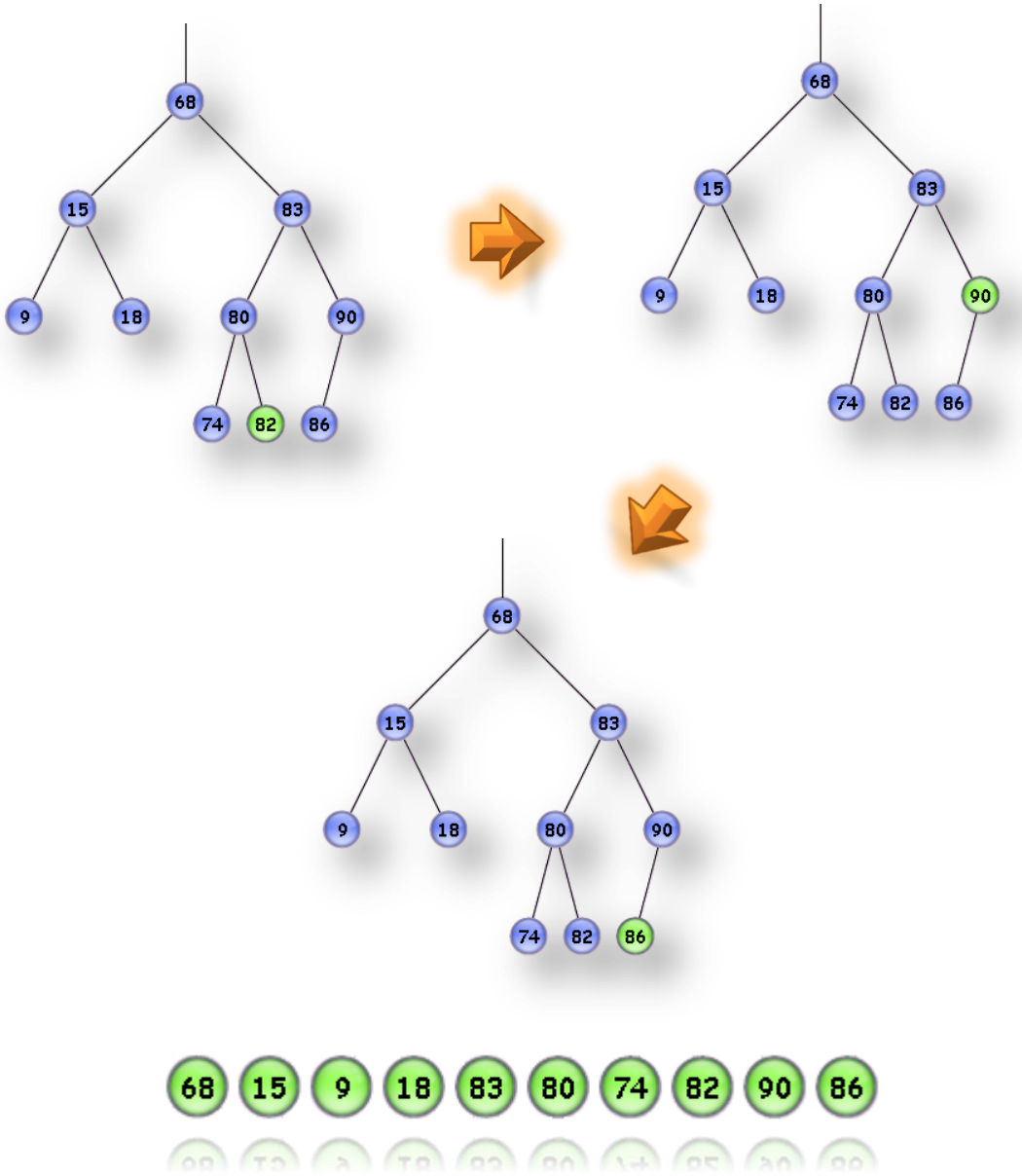
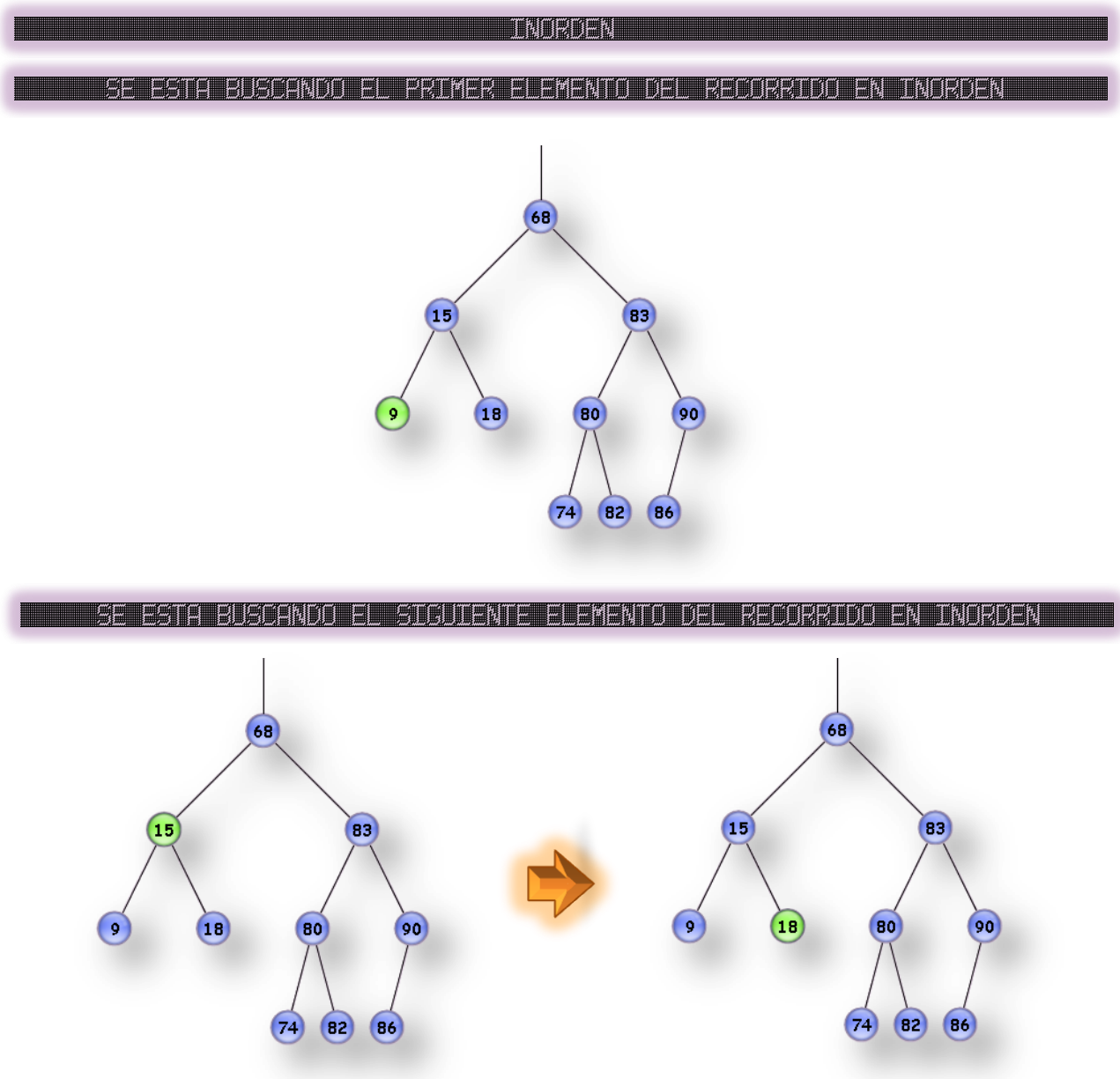
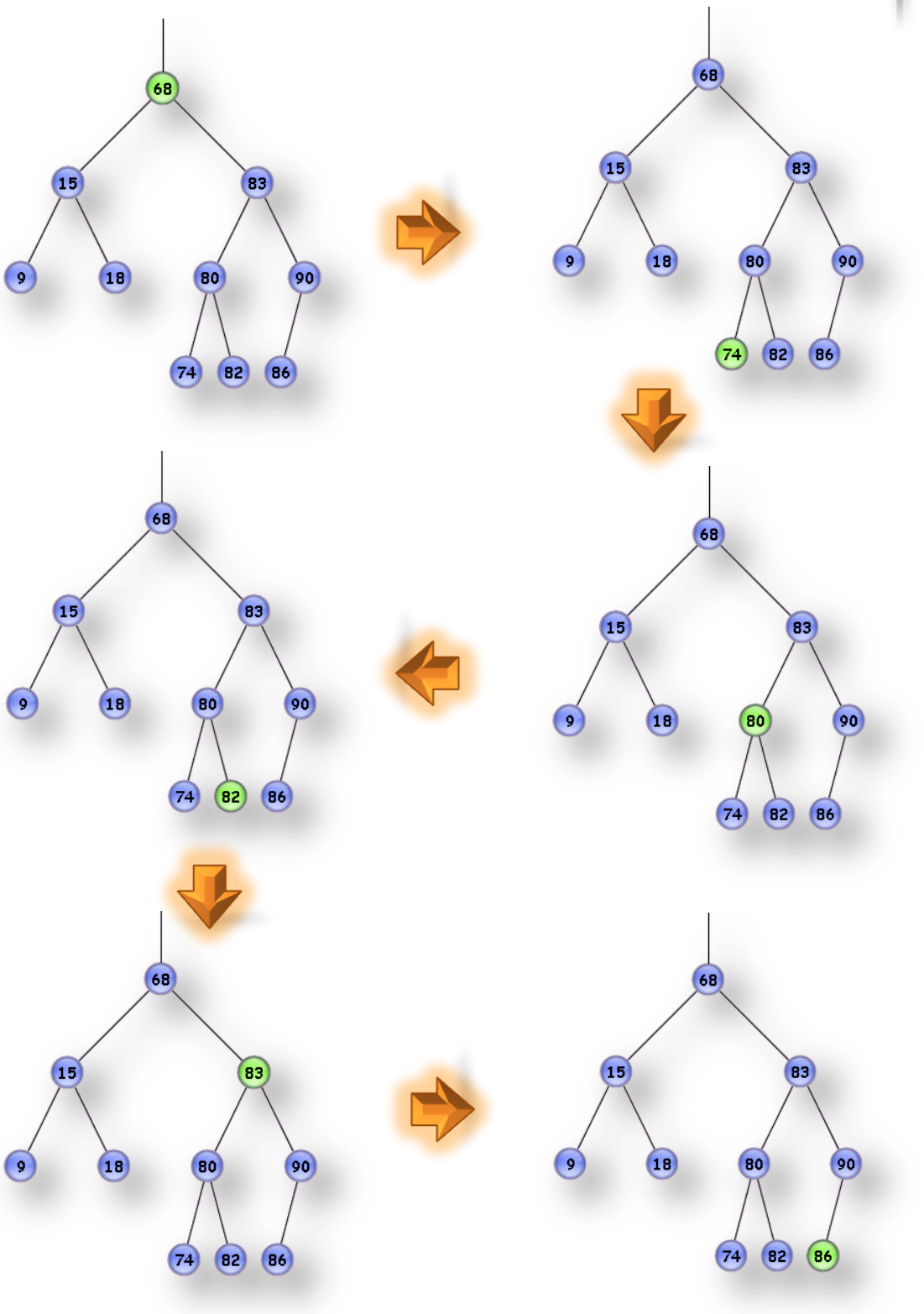


Figura 83 – Animación de la operación Preorden. ABB. Visualización de Usuario.

- ***Inorden***: *al seleccionar esta operación se mostrará, con ayuda de un elemento auxiliar, el recorrido en inorden del árbol. El elemento auxiliar se desplazará de nodo en nodo en un recorrido en inorden. (Ver* ***figura 84***)




68
15
83
9
18
80
90
74
82
86

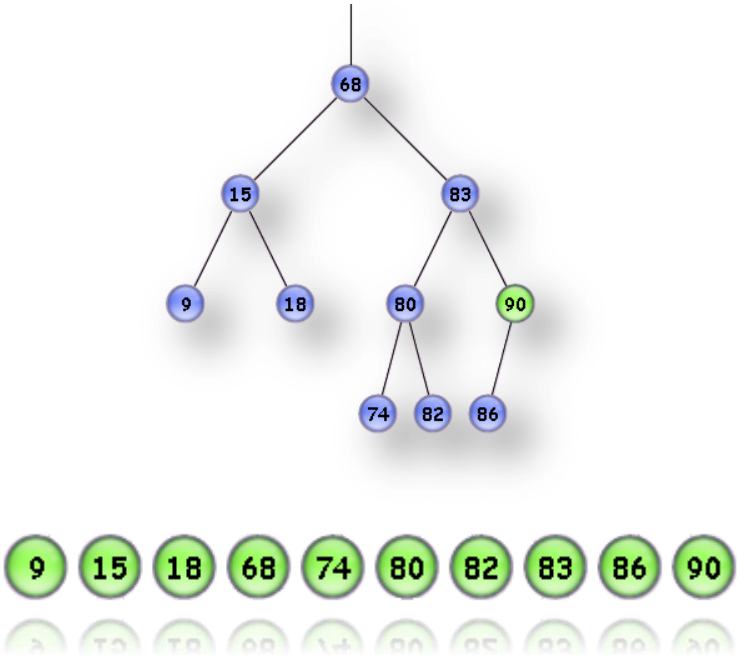
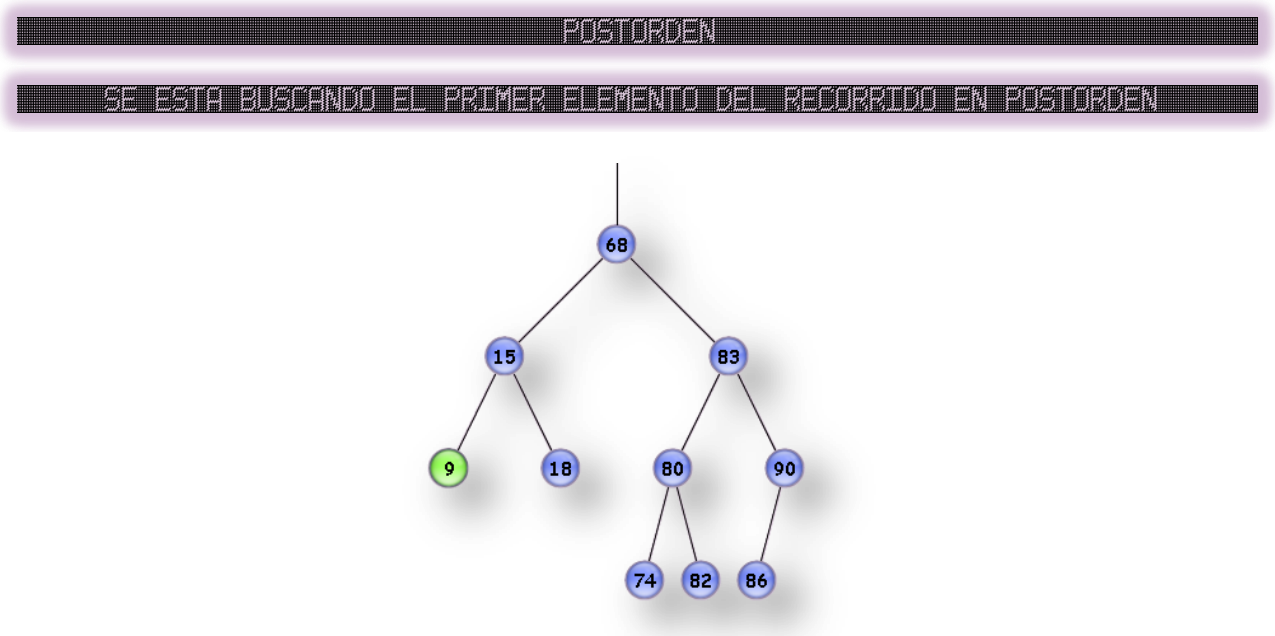
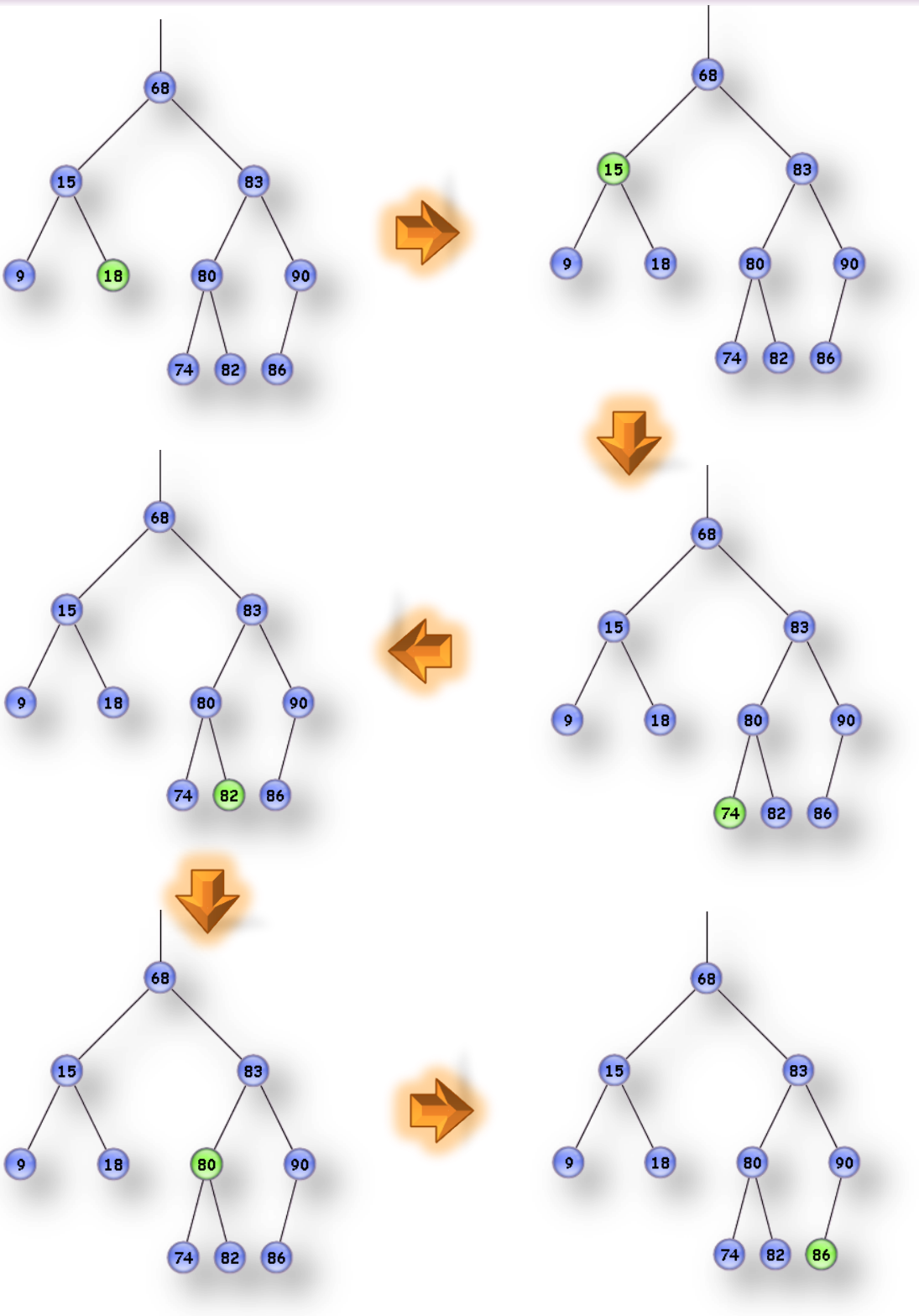


Figura 84 – Animación de la operación Inorden. ABB. Visualización de Usuario.

- **Postorden**: al seleccionar esta operación se mostrará, con ayuda de un elemento auxiliar, el recorrido en postorden del árbol. El elemento auxiliar se desplazará de nodo en nodo en un recorrido en postorden. (Ver **figura 85**)



SE ESTA BUSCANDO EL SIGUIENTE ELEMENTO DEL RECORRIDO EN POSTORDEN



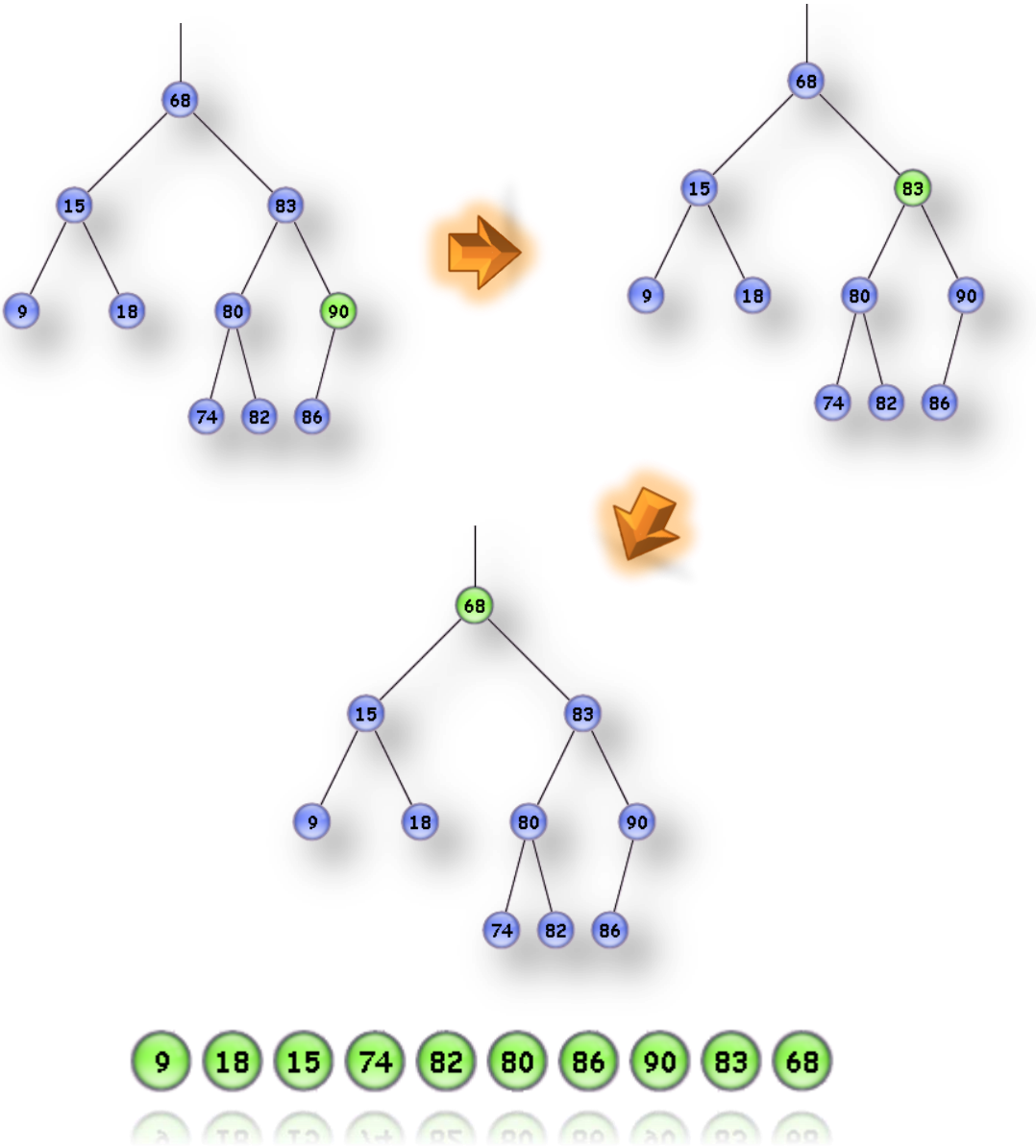
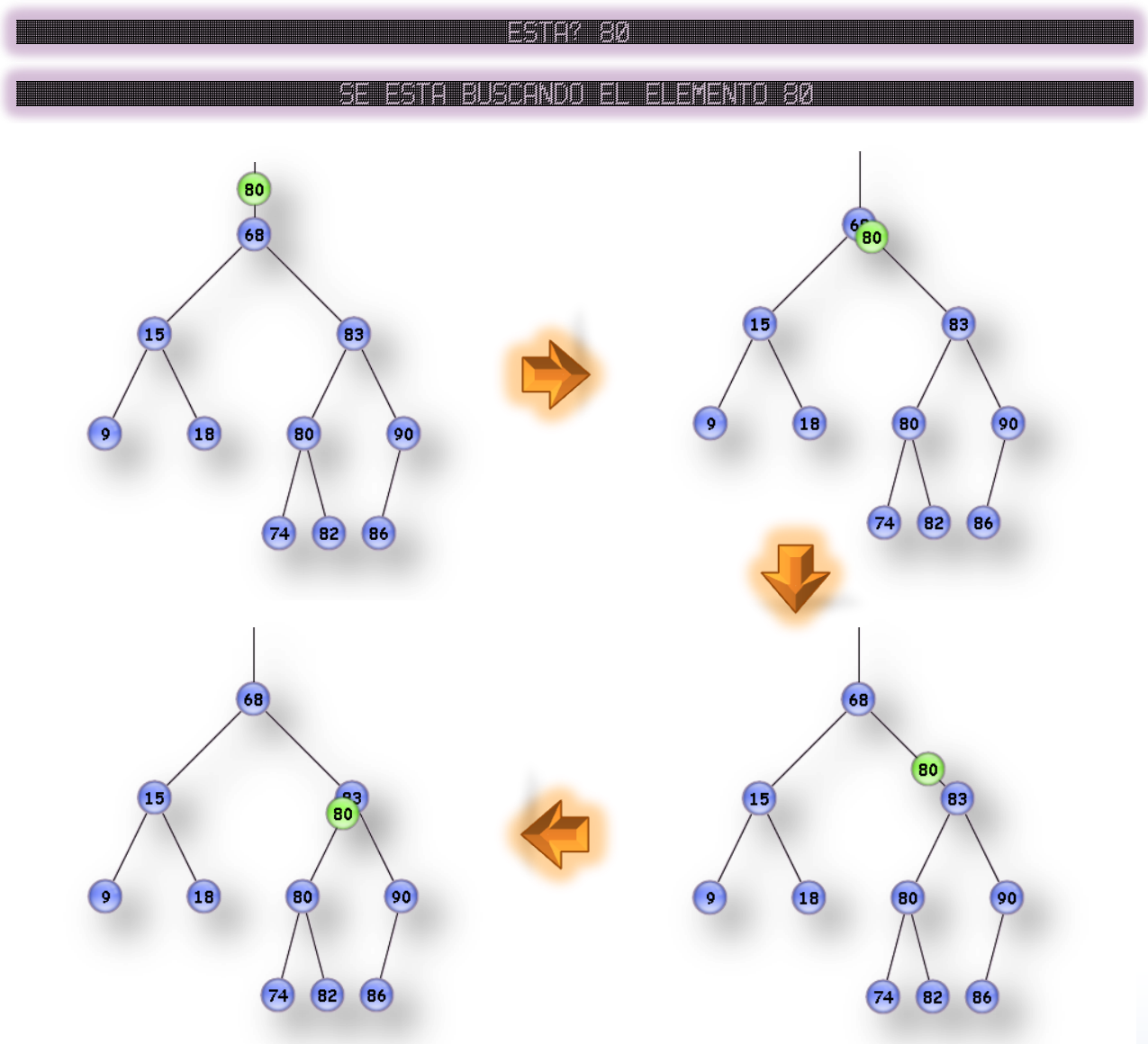


Figura 85 – *Animación de la operación Postorden. ABB. Visualización de Usuario.*

- **Está?:** *al seleccionar esta operación se mostrará la búsqueda del elemento en el árbol, como se muestra en las* **figuras 86** y **87**.



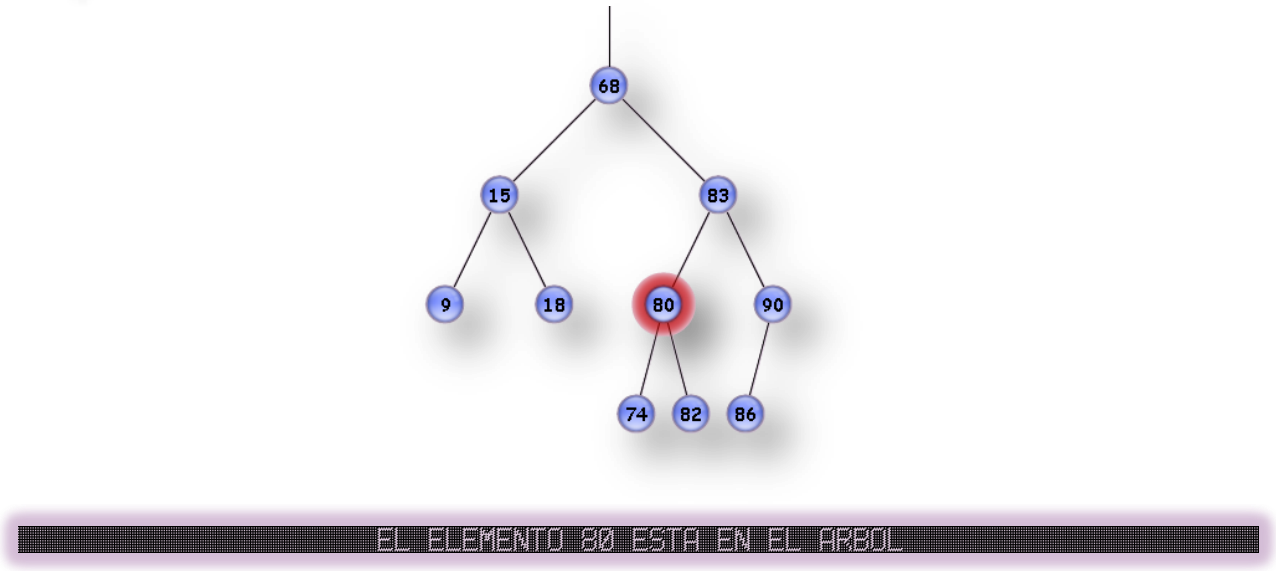
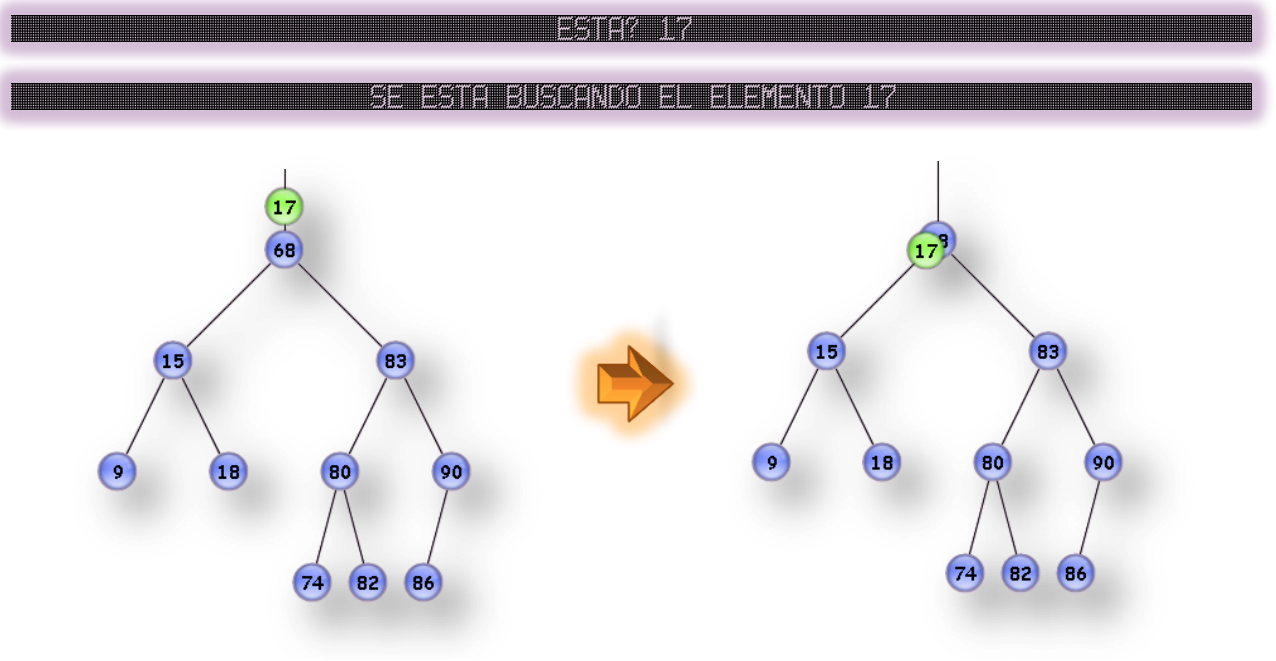


Figura 86 – *Animación de la operación Esta? el elemento 80. ABB. Visualización de Usuario.*



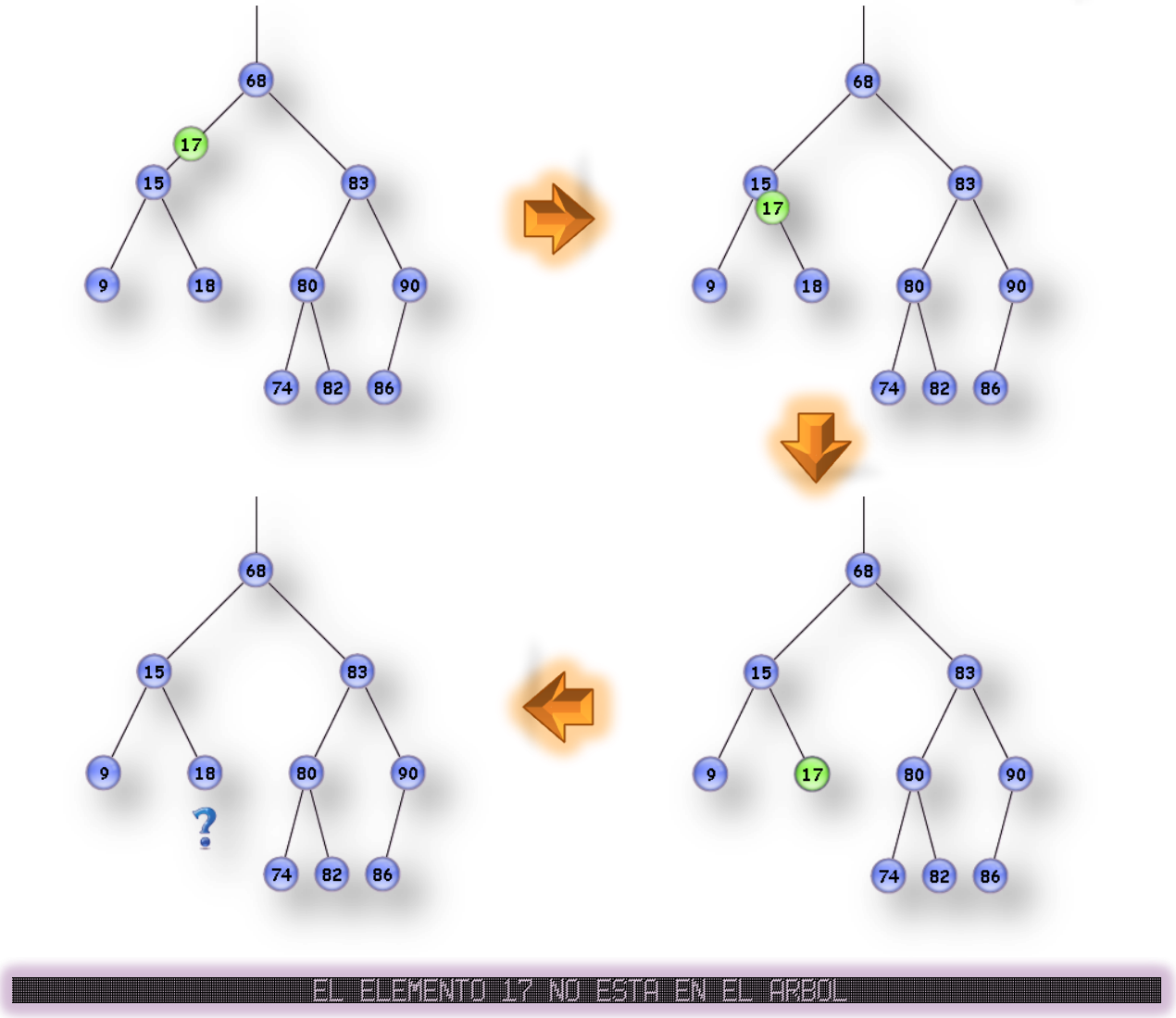
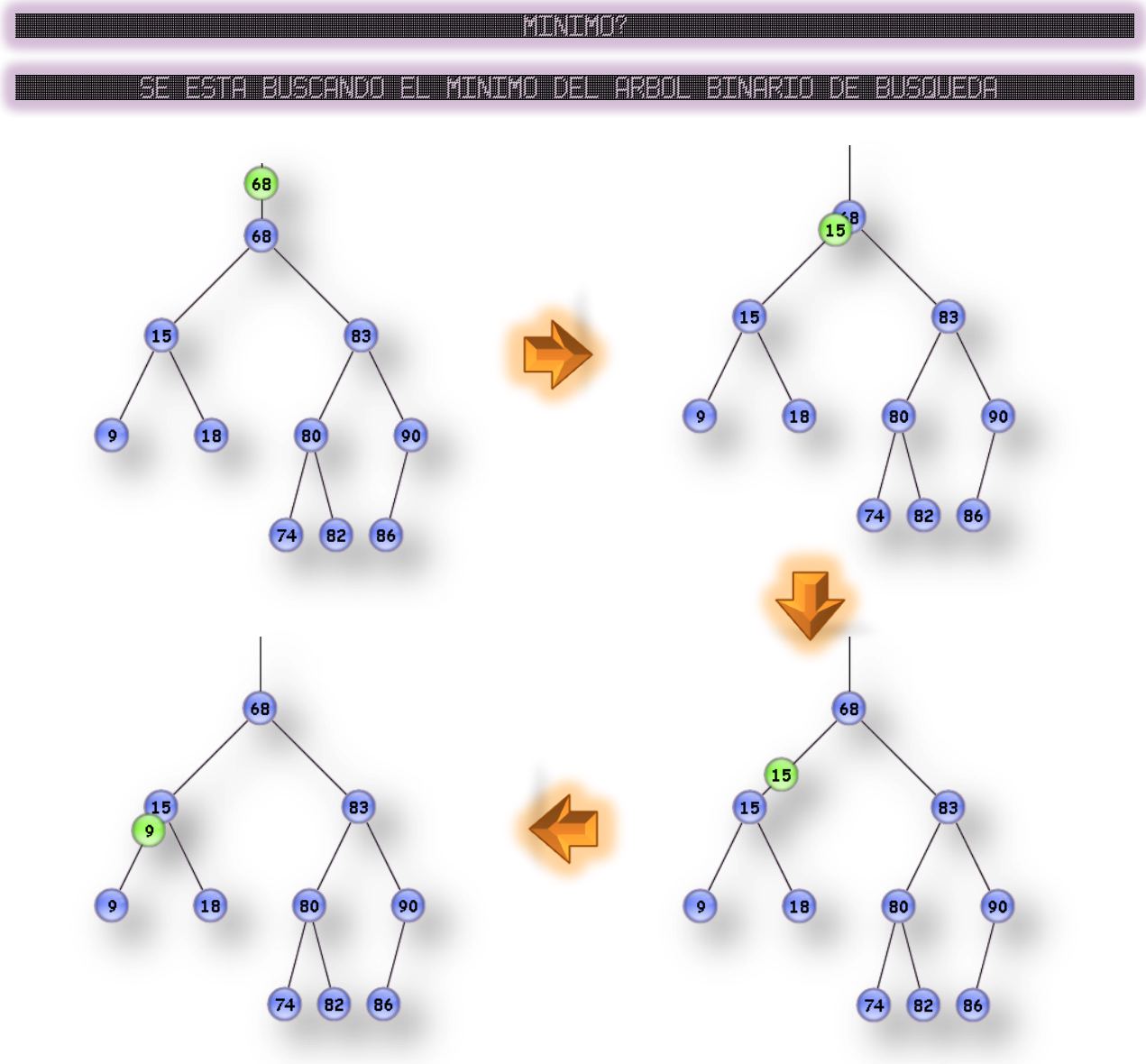


Figura 87 – Animación de la operación Esta? el elemento 17. ABB. Visualización de Usuario.

- **Mínimo?**: *al seleccionar esta operación se mostrará como un elemento auxiliar recorre el árbol en busca del elemento mínimo. (Ver* ***figura 88****).*



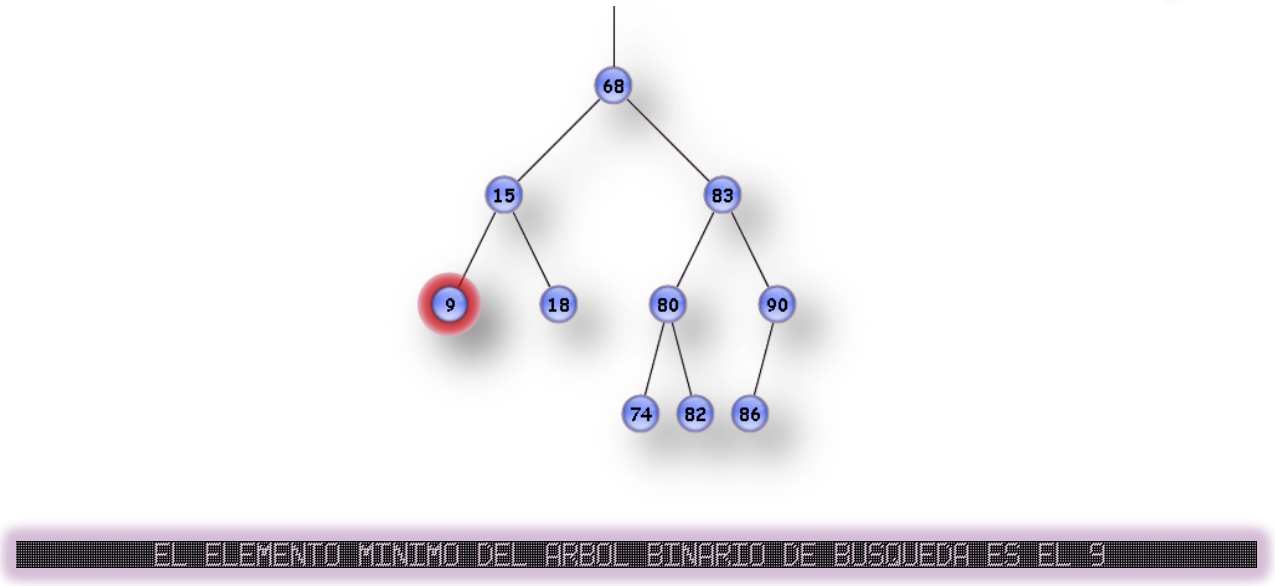
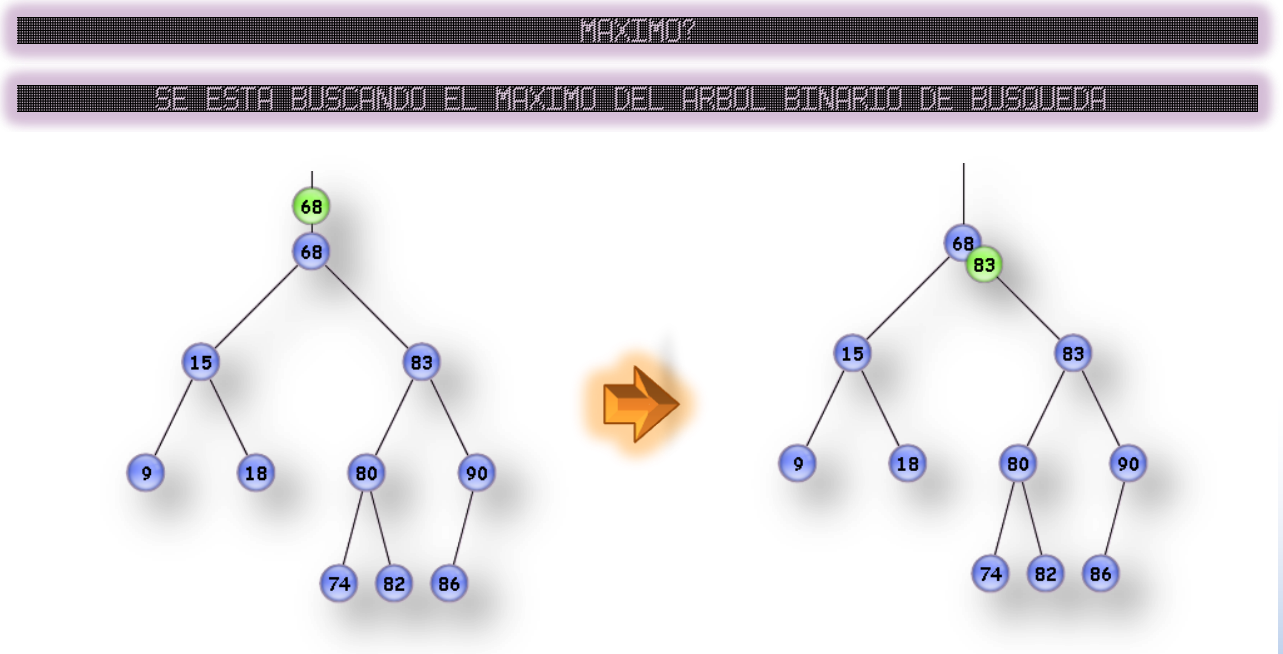


Figura 88 – Animación de la operación Minimo?. ABB. Visualización de Usuario.

- ***Máximo?***: *al seleccionar esta operación se mostrará como un elemento auxiliar recorre el árbol buscando del elemento máximo. (Ver* ***figura 89****).*



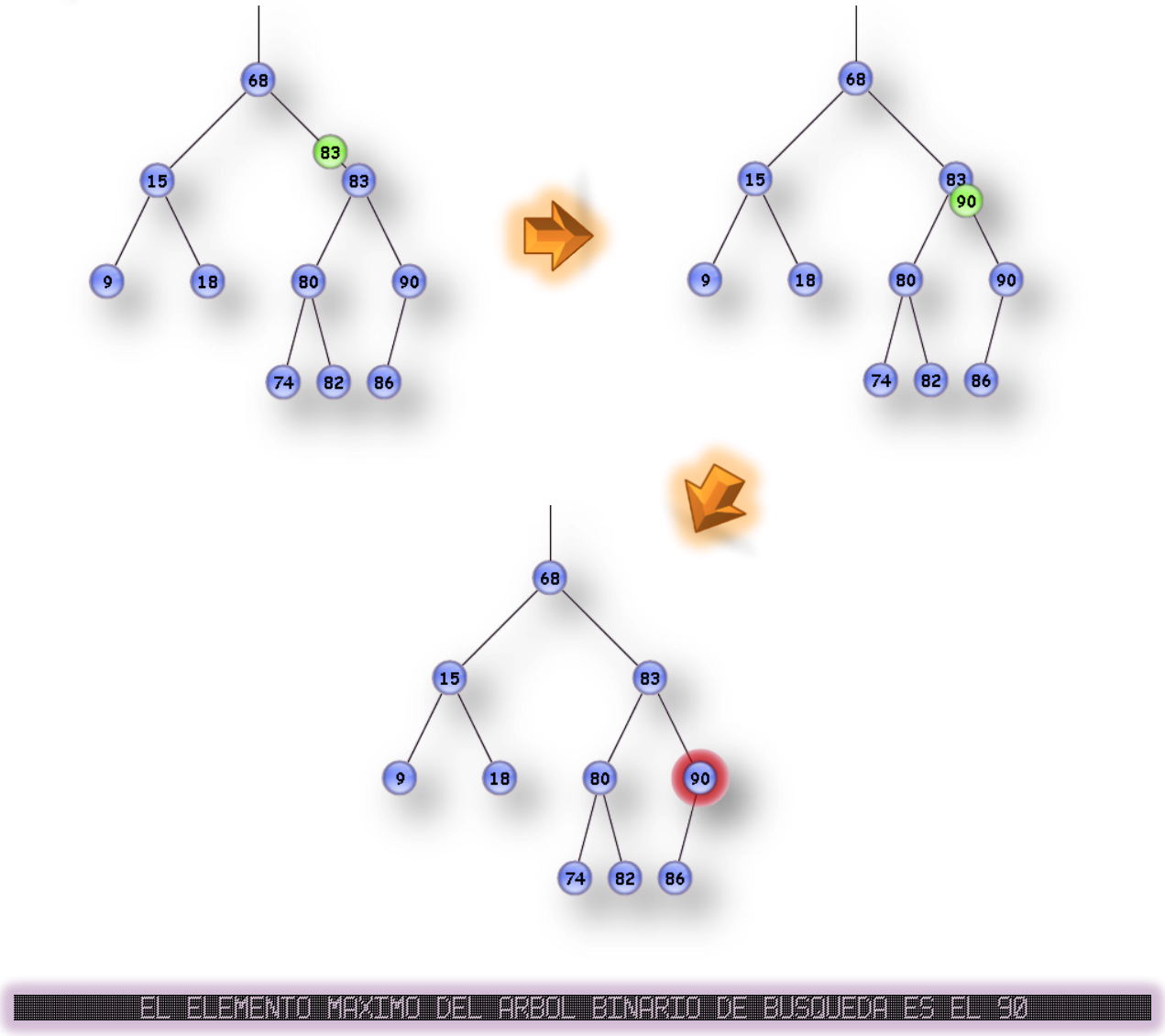
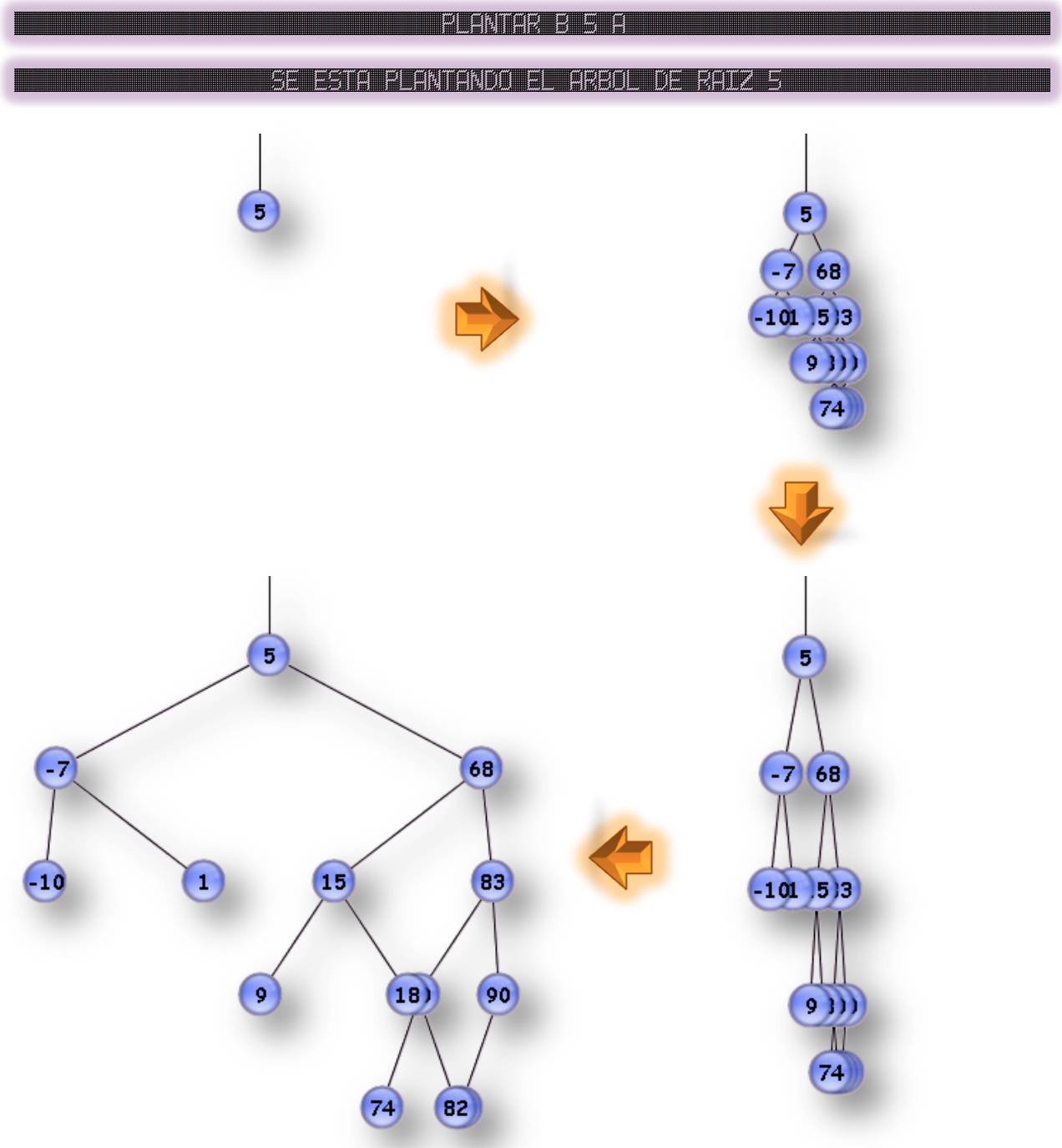


Figura 89 – Animación de la operación Maximo?. ABB. Visualización de Usuario.

- **Plantar**: al seleccionar esta operación se mostrará de forma animada como se planta un árbol binario de búsqueda. Para ello, el usuario tiene que introducir el valor de la raíz, y la aplicación se encarga de controlar qué árbol, de entre los que están abiertos, puede ser el hijo izquierdo y cuál el hijo derecho (este control se realiza comparando el valor de la raíz con el mínimo y el máximo de cada árbol).

Primero aparecerá la raíz y a continuación se desplegaran sus nuevos hijos. Esta secuencia la podemos ver en la **figura 90**, donde hemos plantado la raíz 5 con un hijo izquierdo que hemos creado con las operaciones descritas anteriormente, y como hijo derecho hemos usado el árbol que se ha ido construyendo como ejemplo.



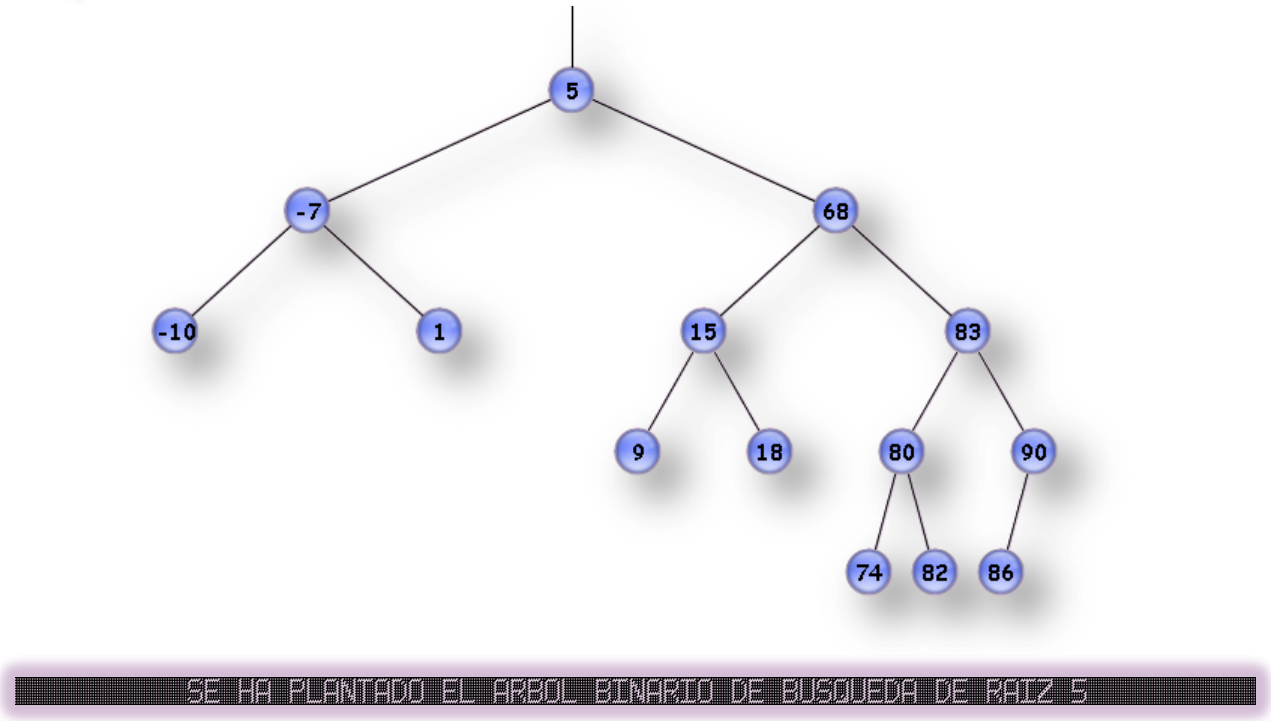


Figura 90 – Animación de la operación Plantar. ABB. Visualización de Usuario.

Como ya se ha comentado, a la operación plantar se le ha asociado dos botones (ver **figura 91**) para mostrar su hijo izquierdo e hijo derecho. Si pulsamos sobre ellos se desplegarán unas ventanas, como las de las **figuras 92** y **93**.

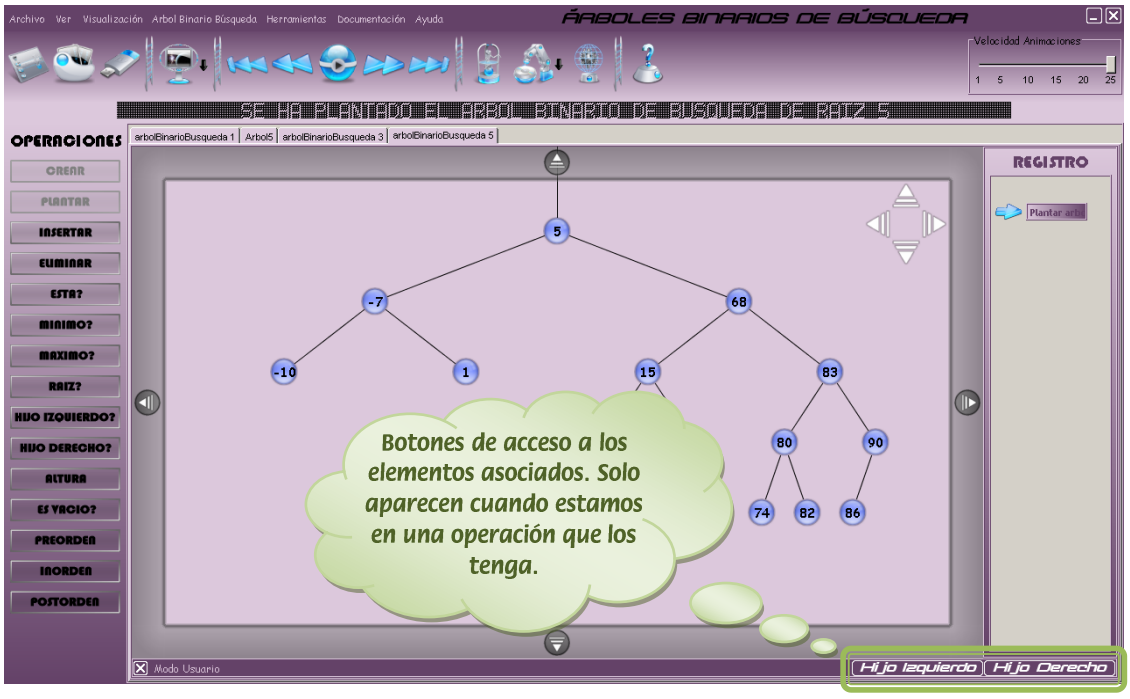


Figura 91 – Botones de acceso a los elementos asociados. ABB. Visualización de Usuario.

Figura 92 – Elemento asociado a la operación Plantar, HijoIzquierdo. ABB. Visualización de Usuario.

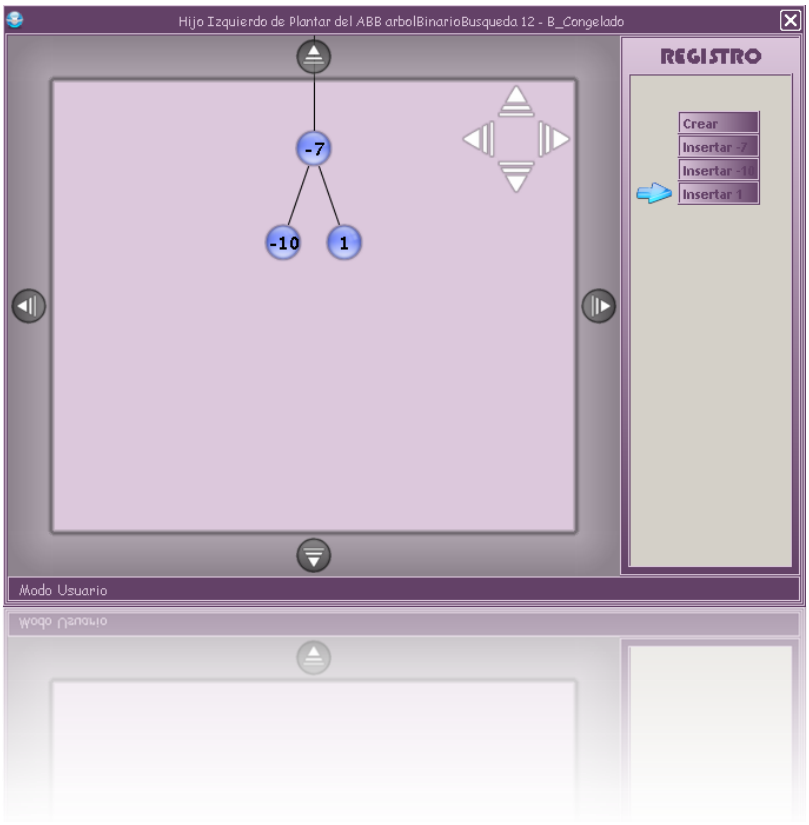
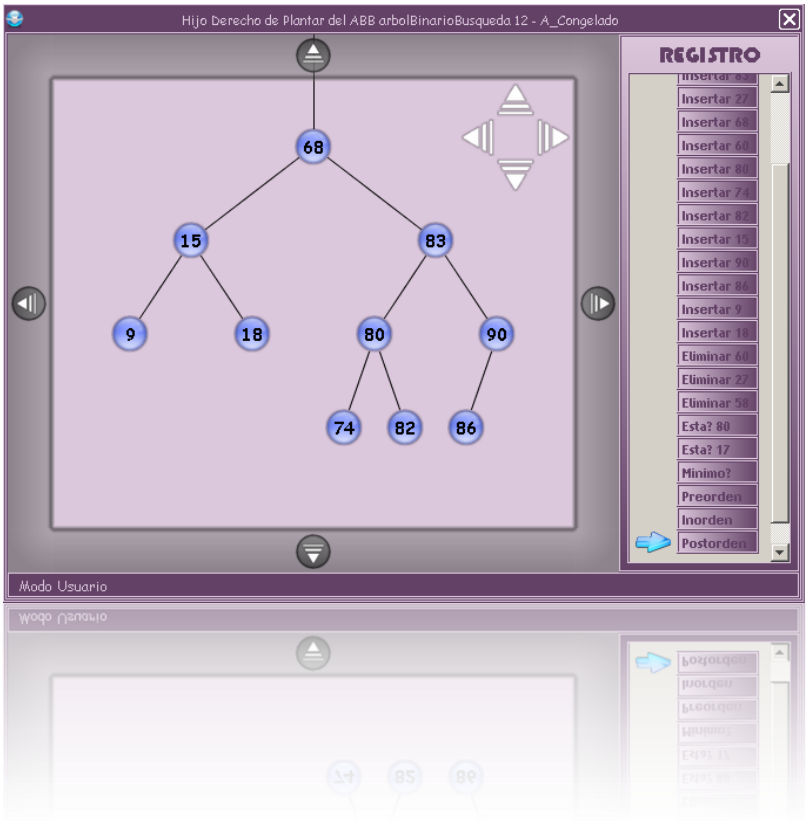


Figura 933 – Elemento asociado a la operación Plantar, HijoDerecho. ABB. Visualización de Usuario.



3.4.- Funcionalidad de la Ventana Estructura de Datos

Como se ha comentado en la introducción, a esta nueva versión de **VEDYA** se le ha dotado de un gran abanico de opciones. Hagamos a continuación un recorrido por el menú de la aplicación, donde están accesibles cada una de estas nuevas características:

3.4.1.- Menú Archivo

Como en cualquier aplicación que se precie, **VEDYA** tiene el menú archivo para la gestión de la entrada y salida de la aplicación. Las opciones que nos permite este menú son (ver ***figura 94*** y ***tabla 2***):

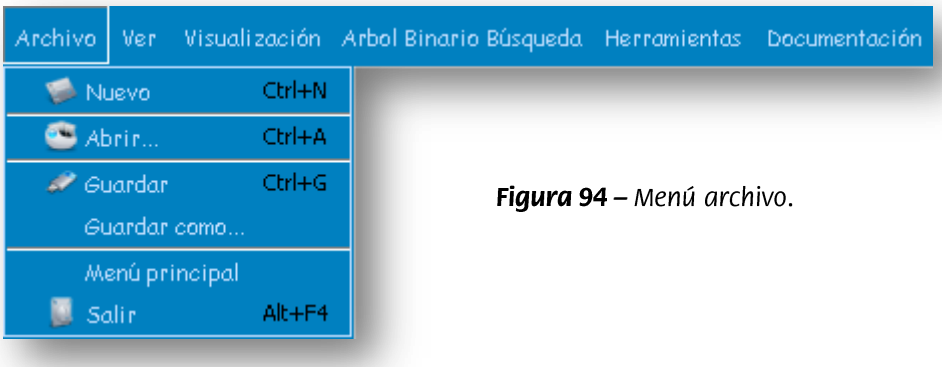


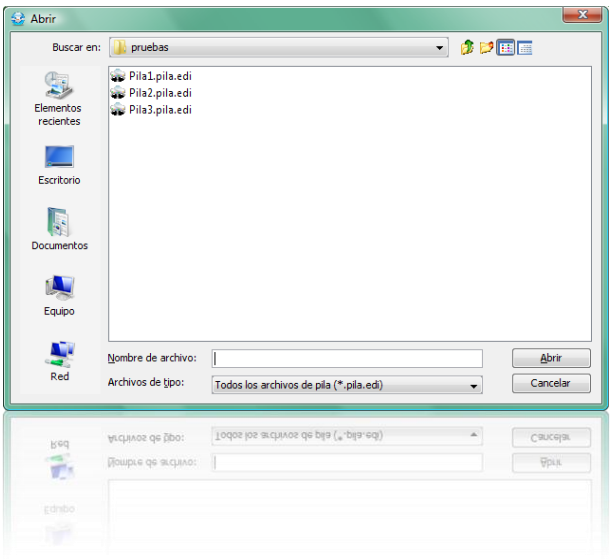
Figura 94 – Menú archivo.

Nuevo	Abrir	Guardar	Salir
Creamos una nueva pestaña.	Abrimos una simulación existente..	Guardamos una simulación en un archivo.	Cerramos la ventana.

Tabla 2 – Controles de archivo

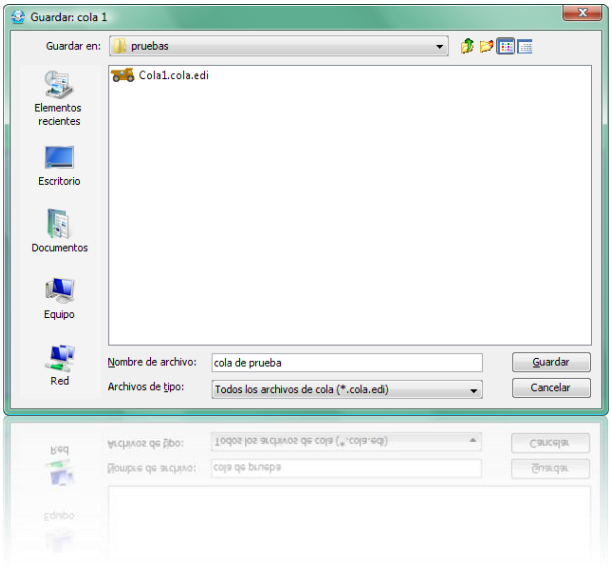
- **Nuevo**: crea una pestaña en blanco en la que se puede crear una nueva simulación de una estructura de datos del tipo de la ventana. (Esta funcionalidad también es accesible desde la barra de herramientas)
- **Abrir**: permite abrir una simulación de una estructura de datos. Para ello, se abre una ventana donde el usuario puede elegir un archivo que contenga una simulación de una estructura de datos. A los archivos que contienen simulaciones de estructuras de datos, se les ha asociado extensiones de archivo propias a cada estructura, así, por ejemplo, las simulaciones de las pilas se guardarán en archivos "***.pila.edi**". Para intentar evitar errores, se ha forzado a que desde una ventana de una cierta estructura de datos solo se puedan abrir archivos con la extensión de esa estructura (ver **figura 95**). (Esta funcionalidad también es accesible desde la barra de herramientas)

Figura 95 – Ventana abrir archivo.



- **Guardar**: guarda la simulación de la estructura de datos contenida en la pestaña seleccionada. Si la simulación tiene un archivo asociado se guarda sin pedir sin ningún tipo de confirmación. En otro caso, la herramienta pregunta al usuario el nombre y la ubicación del archivo donde quiere almacenar la simulación, mostrando una ventana como la de la **figura 96**. (Esta funcionalidad también es accesible desde la barra de herramientas).

Figura 96 – Ventana guardar archivo.



- ***Guardar como***: guarda la simulación de la estructura de datos contenida en la pestaña seleccionada en el archivo y la ubicación seleccionados por el usuario en una ventana como la de la **figura 96**.
- ***Menú principal***: minimiza la ventana actual y maximiza la ventana principal.
- ***Salir***: cierra la ventana actual. Las acciones derivadas de esta acción se comentan al final de esta sección.

3.4.2.- Menú Ver

Este menú nos da la posibilidad de ocultar o mostrar determinados elementos de la interfaz. Las opciones que nos permite este menú son (ver **figura 97**):

Figura 97 – Menú ver.

- ***Barra de herramientas***: *menú que contiene botones para mostrar u ocultar los accesos rápidos a los menús de "Archivo", "Visualización", "Simulación", "Documentación" y "Ayuda", teniendo la posibilidad de ocultar todo o mostrarlo. (Ver* ***figuras* 98**, **99** y **103**)
- ***Cuadro de diálogo***: *oculta o muestra el cuadro de diálogo de la aplicación, donde se muestran comentarios acerca del funcionamiento de la aplicación. (Ver* ***figuras* 98**, **100** y **103**)
- ***Caja de operaciones***: *oculta o muestra la caja de operaciones situada en la parte izquierda de la ventana. Esta caja, contiene los botones de acceso rápido para la ejecución de las operaciones de la estructura. (Ver* ***figuras* 98**, **101** y **103**)
- ***Registro de operaciones***: *oculta o muestra el registro de operaciones situado en la parte derecha de la ventana. Este registro, contiene el listado de todas las operaciones ejecutadas en la simulación de la pestaña. (Ver* ***figuras* 98**, **102** y **103**)

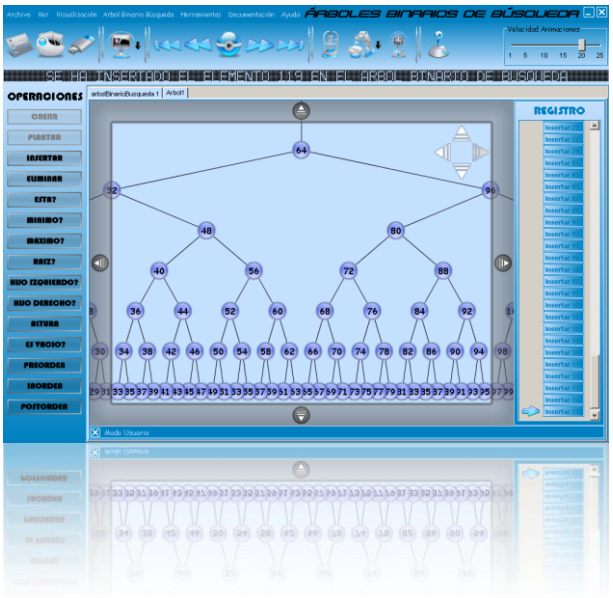


Figura 98 – *Ventana normal.*

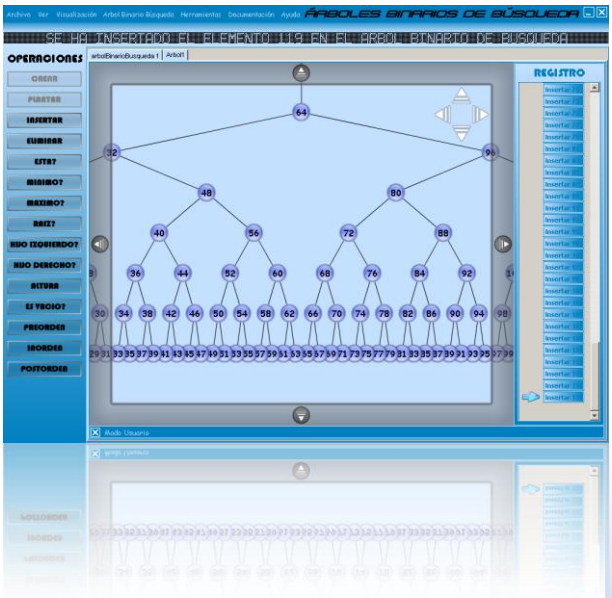


Figura 99 – *Ventana sin barra de herramientas.*

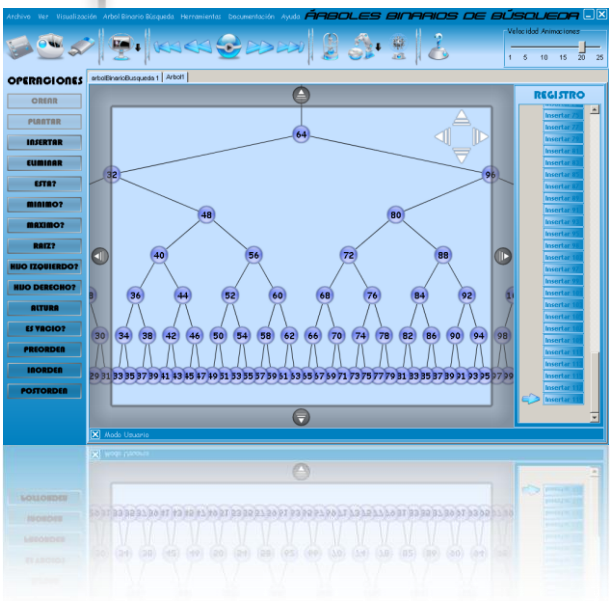


Figura 100 – *Ventana sin cuadro de diálogo.*

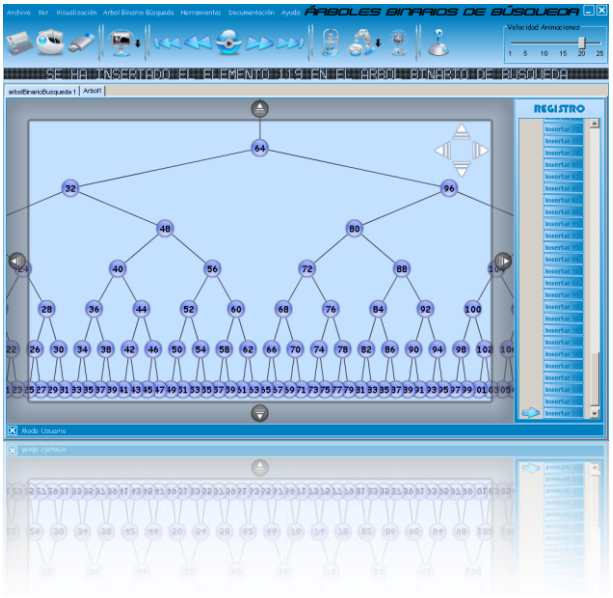


Figura 101 – *Ventana sin caja de operaciones.*

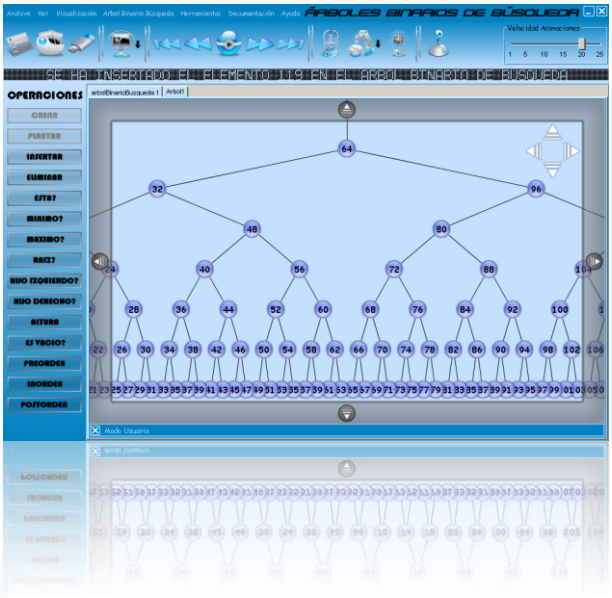


Figura 1022 – *Ventana sin registro de operaciones.*

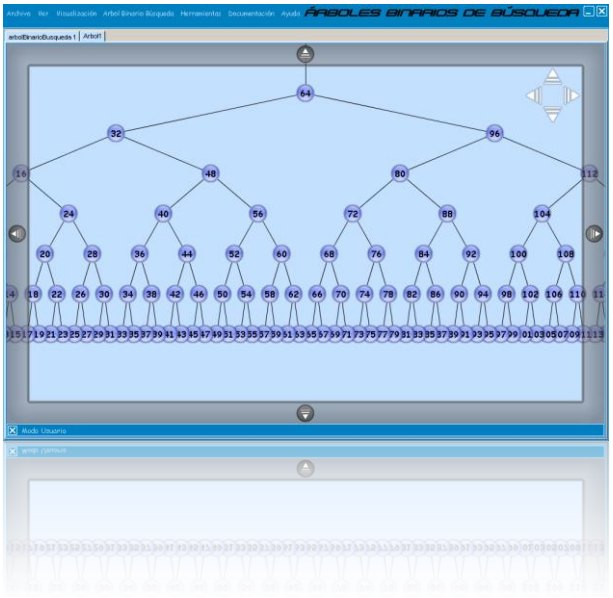


Figura 1033 – *Ventana sin nada.*

3.4.3.- Menú Visualización

Este menú nos permite cambiar el modo de visualización de la pestaña seleccionada, para comprender las estructuras desde diferentes puntos de vista (ver **figura 104**). Esta funcionalidad, que también es accesible desde la barra de herramientas, suele contener las siguientes visualizaciones (estás pueden variar según la estructura):

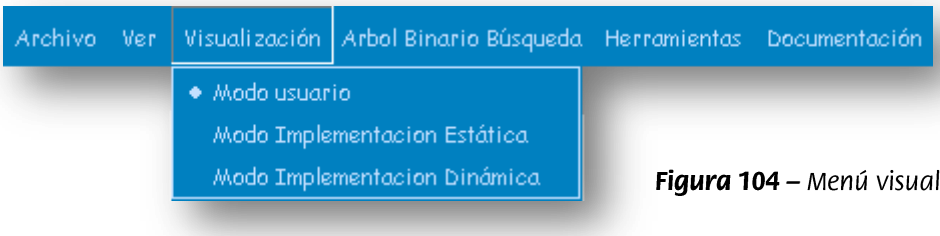


Figura 104 – Menú visualización.

- **Modo usuario**: modo en el cual se muestran, de forma original, las principales características de la estructura de datos
- **Modo Implementación Estática**: modo en el cuál se muestra el comportamiento de la estructura mediante una representación estática.
- **Modo Implementación Dinámica**: modo en el cuál se muestra el comportamiento de la estructura mediante una representación dinámica.

3.4.4.- Menú Estructura

Cada ventana tiene un menú específico con las operaciones propias de la estructura de datos. Estas operaciones son las que están disponibles en la caja de operaciones y las que se han especificado en la **sección 3.3.** (Ver **figuras 105**, **106** y **107**)

Figura 105 – Menú con las operaciones de la pila.

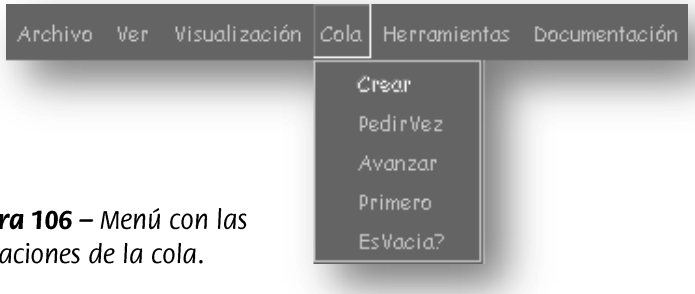


Figura 106 – *Menú con las operaciones de la cola.*

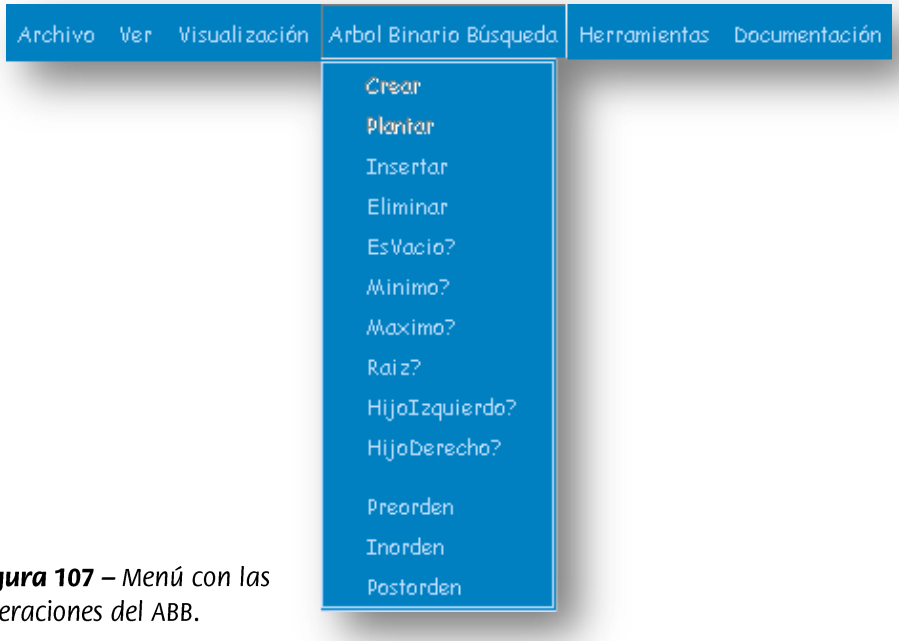


Figura 107 – *Menú con las operaciones del ABB.*

3.4.5.- Menú Herramientas

Menú que nos permite configurar ciertos aspectos de la aplicación: (ver ***figura 108****)*

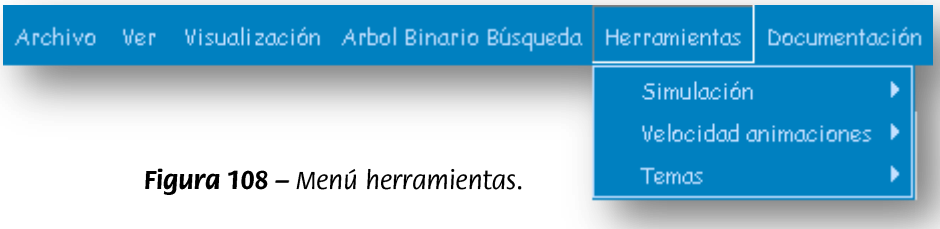


Figura 108 – *Menú herramientas.*

- **Simulación**: *submenú que nos permite acceder a los controles de simulación de la aplicación. (Ver* ***figura* 109**)

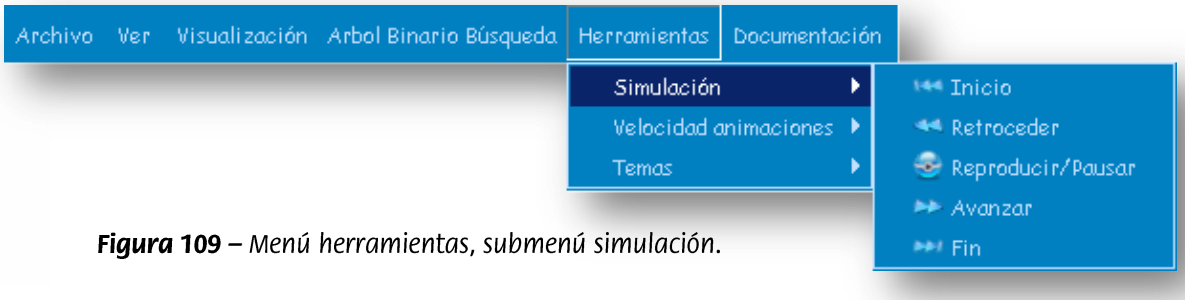


Figura 109 – *Menú herramientas, submenú simulación.*

Al pulsar sobre cualquier operación de una estructura de datos, se visualiza la animación correspondiente a dicha operación. Esta operación es añadida al registro de operaciones, situada en la parte derecha de la pestaña. Esta es la forma de simulación más básica, y única, que se había implementado hasta ahora.

Una de las novedades que hemos añadido a **VEDYA**, *ha sido la posibilidad de controlar la simulación de las operaciones realizadas sobre una estructura de datos, mediante los controles que aparecen en el menú y en la barra de herramientas (ver* ***figura* 110**).

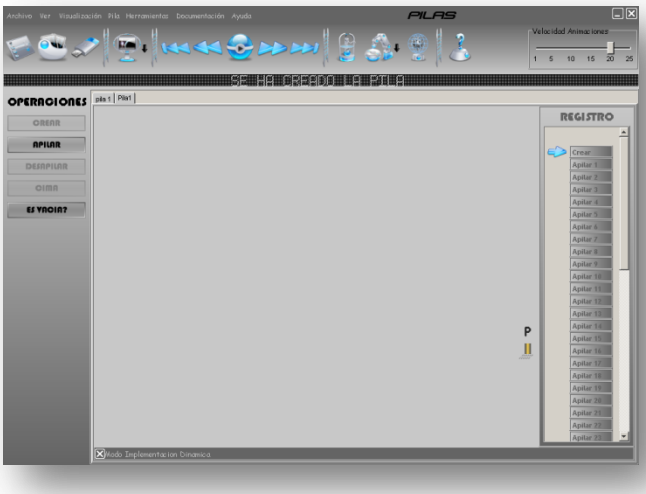
En la ***tabla* 3** *se explican los controles de simulación disponibles:*

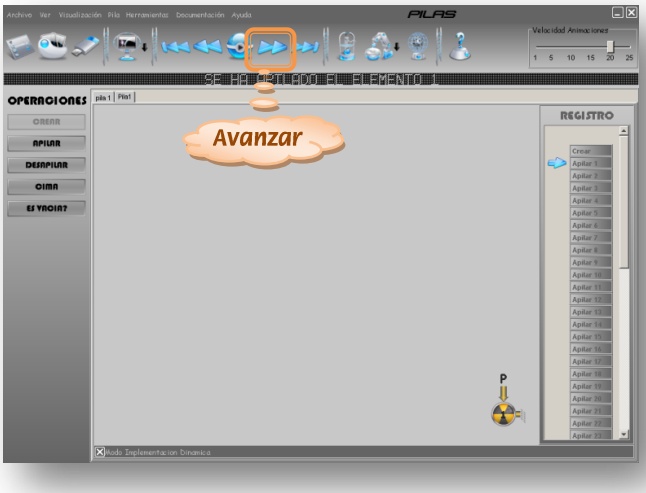
Inicio	Retroceder	Play/Stop	Avanzar	Fin
Retrocedemos a la primera operación de la simulación.	Retrocedemos a la operación anterior.	Iniciamos/Paramos la simulación de las operaciones de una estructura de datos	Avanzamos a la operación siguiente.	Avanzamos a la última operación de la simulación.

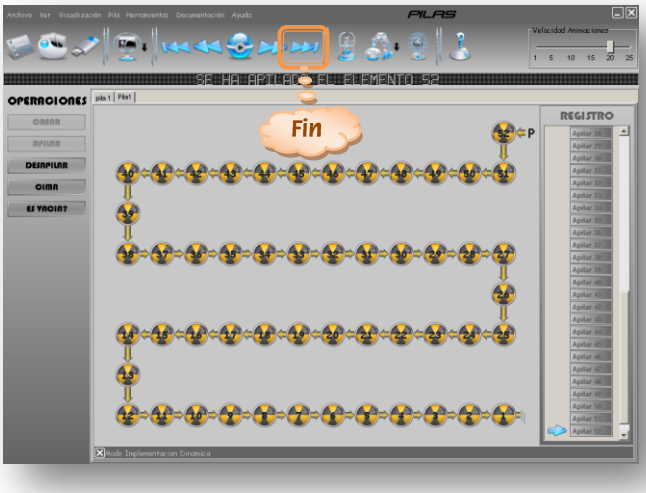
Tabla 3 – *Controles de simulación*

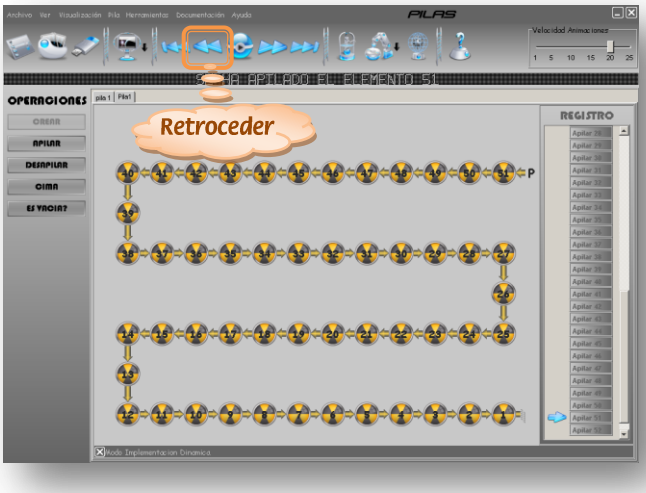
Además, también se puede controlar la simulación en el registro de operaciones, de forma que pinchando sobre una operación en concreto vamos directamente al estado de la estructura en el cual la última operación ejecutada fue la operación elegida.

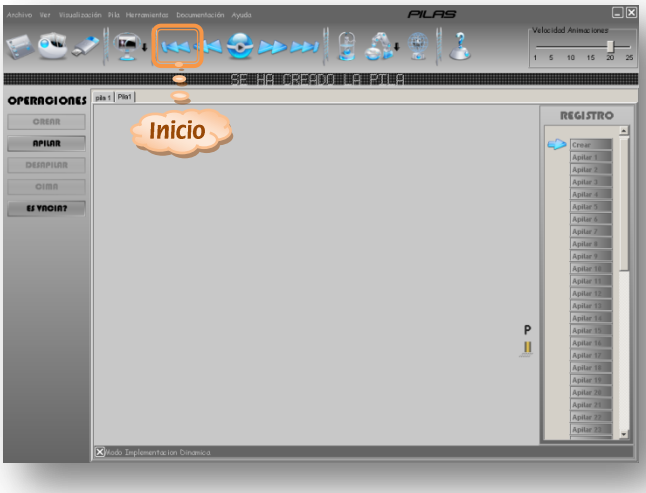
Este control sobre la simulación, además de permitirnos ver una y otra vez una misma simulación sin tener que crearla de nuevo, está capacitado para alterar las simulaciones en cualquier punto. Es decir, si nos situamos en cualquier operación, y pulsamos una operación de la estructura de datos, las operaciones que hubiese hasta el final se eliminan de la simulación.






PILAS
SE HA APILADO EL ELEMENTO 52
Fin
OPERACIONES
REGISTRO


PILAS
SE HA APILADO EL ELEMENTO 51
Retroceder
OPERACIONES
REGISTRO


PILAS
SE HA CREADO LA PILA
Inicio
OPERACIONES
REGISTRO

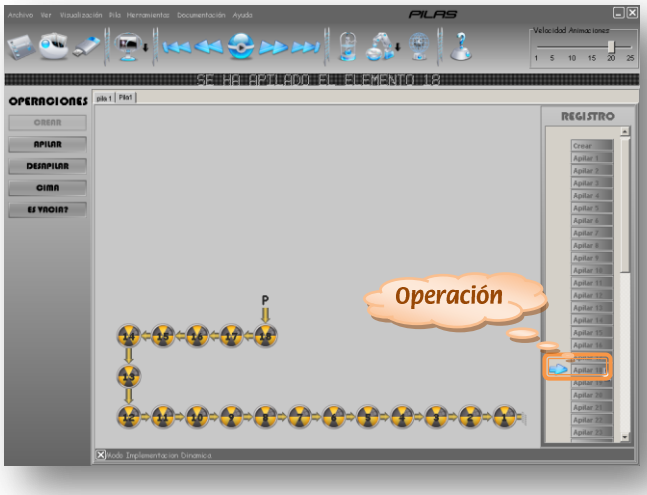


Figura 110 – *Utilización de los controles de simulación.*

- ***Velocidad animaciones***: *permite cambiar la velocidad de las animaciones. Contiene los valores discretos: muy lento, lento, normal, rápido y muy rápido. (Ver* ***figura 111****). Esta funcionalidad también está disponible, para una mayor comodidad para el usuario, en la barra de herramientas.*

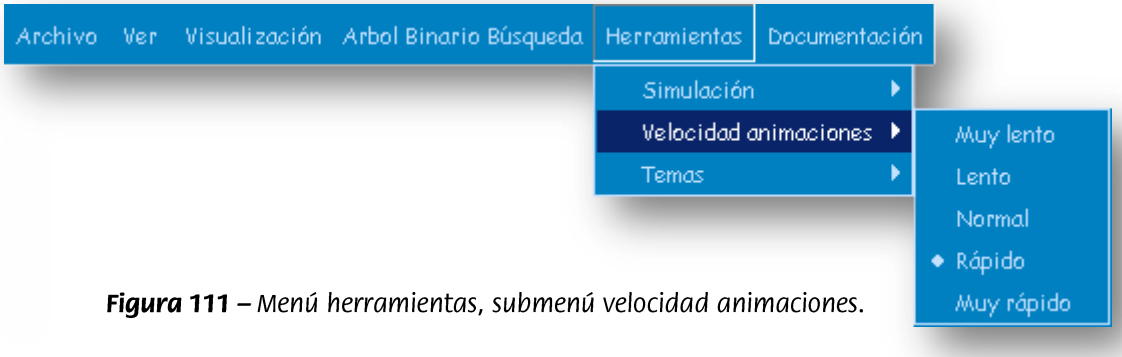


Figura 111 – *Menú herramientas, submenú velocidad animaciones.*

- ***Temas***: *nos permite cambiar los colores de la interfaz de usuario (ver* ***figura 112****). Implementado para intentar conseguir un entorno más amigable para* **VEDYA**.

  Se han elegido las 14 combinaciones de colores como se muestra en la ***figura 113****.*

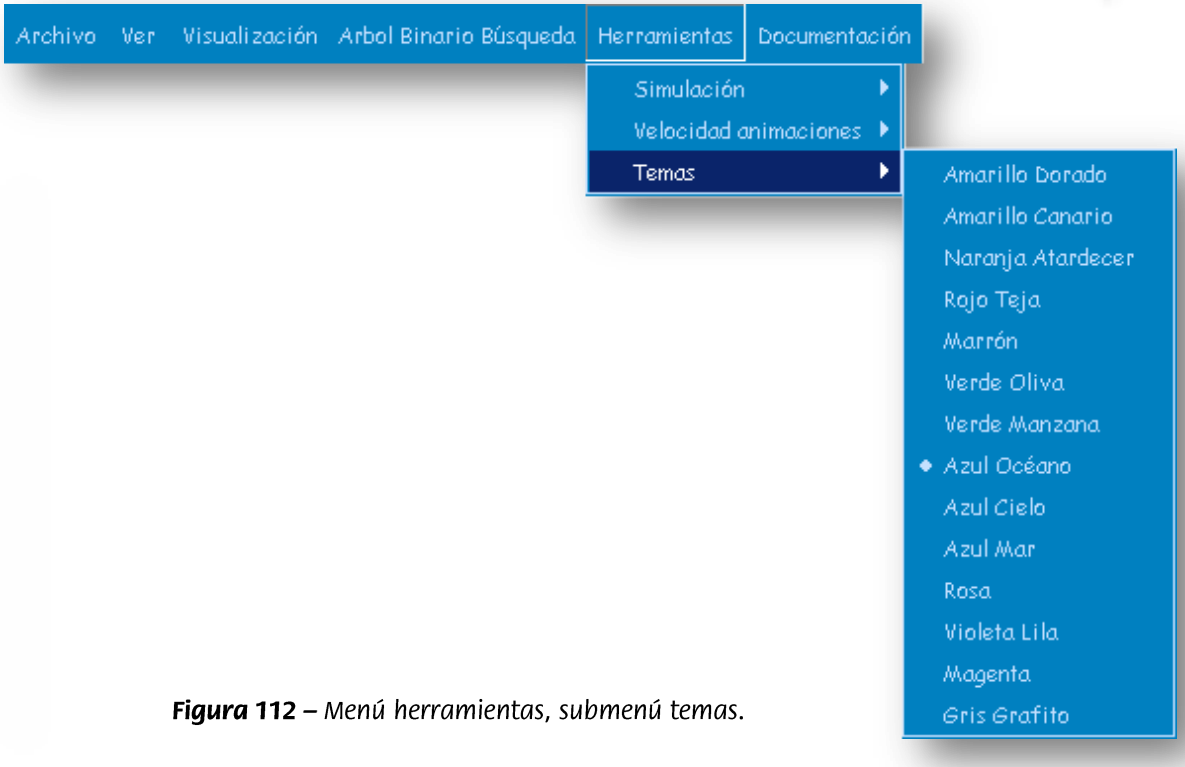


Figura 112 – *Menú herramientas, submenú temas.*

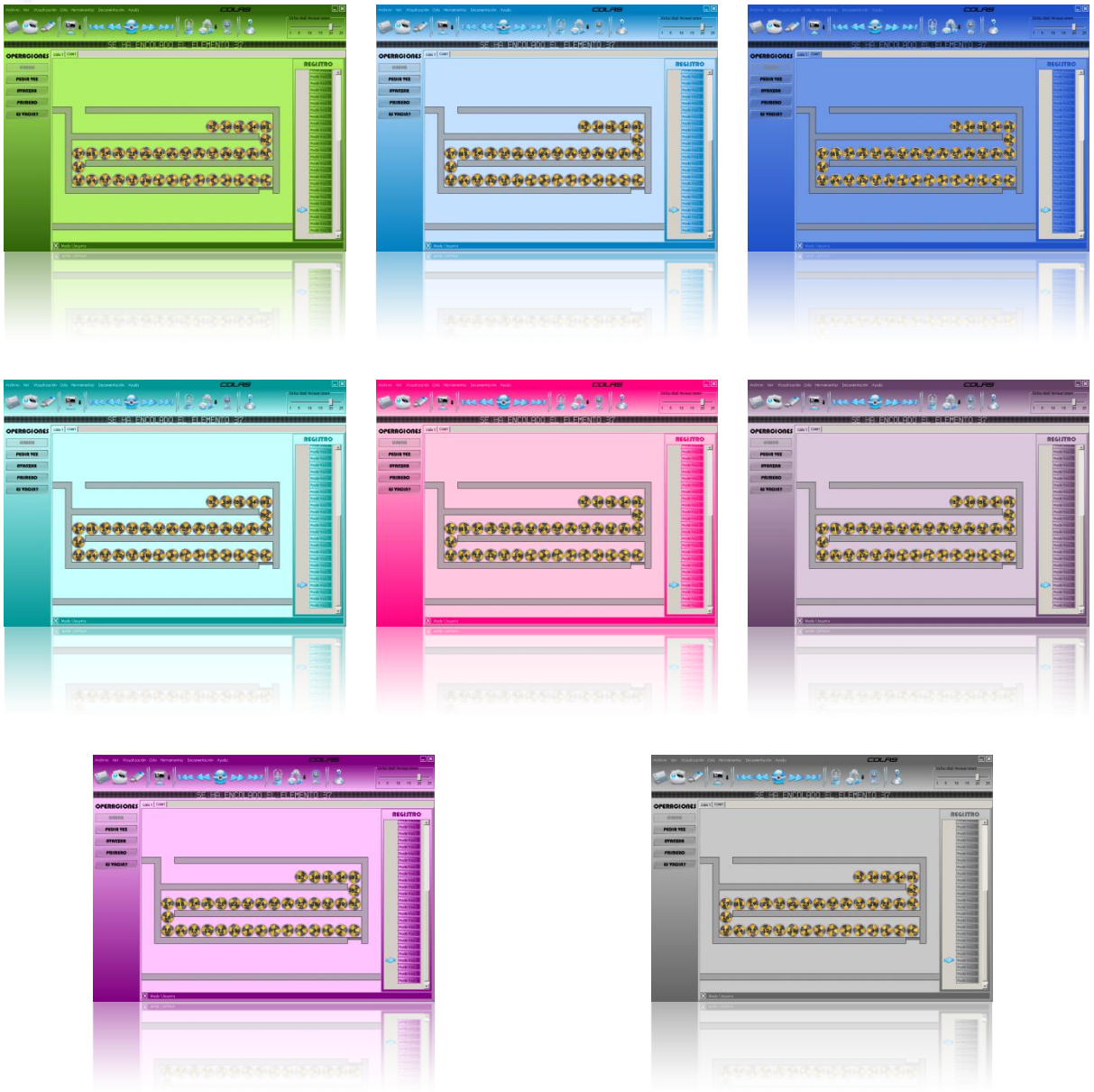


Figura 113 – *Combinaciones de colores de* **VEDYA**.

3.4.6.- Menú Documentación

Las opciones que se despliegan en este menú están relacionadas con la documentación que contiene **VEDYA** *sobre la estructura de datos, que se está tratando, para completar el aprendizaje de la estructura (ver* ***figura 114****). Los submenús que contiene, a excepción de Test que comentamos a continuación, son los que se introdujeron en versiones anteriores de* **VEDYA**.

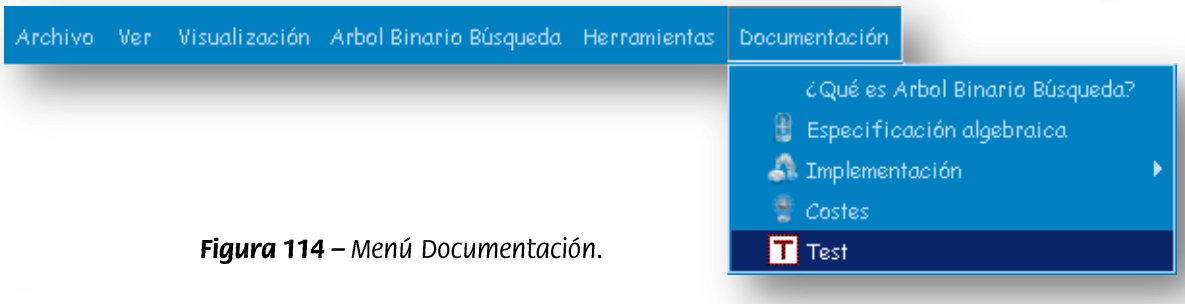


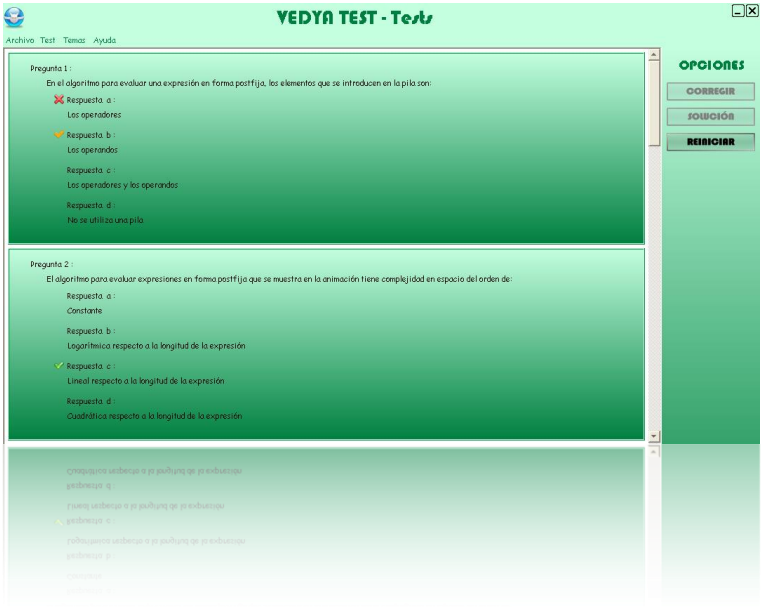
Figura 114 – Menú Documentación.

- **Test**: permite al usuario realizar test para autoevaluar sus conocimientos. Al pulsar sobre este menú se abre una ventana como la que se muestra en la figura. La funcionalidad que se ofrece está encerrada en los menús y en los botones situados en la parte derecha de la ventana. Así podemos:
  - Abrir un test que nos haya dejado el profesor. El test, con todas sus preguntas y respuestas, se desplegará en el panel central. Es ahora, cuando el usuario puede seleccionar las respuestas que crea correctas (ver **figura 115**).

Figura 115 – Ventana que muestra un test preparado para su realización.

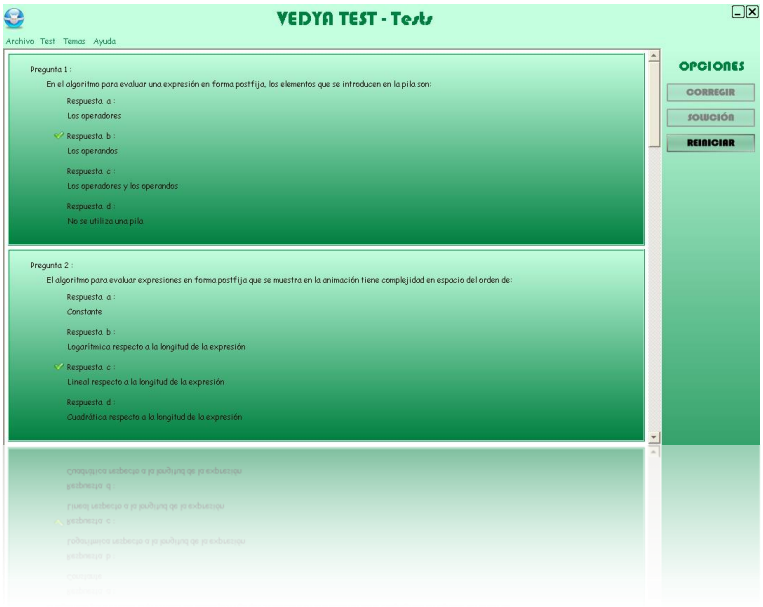
  - Cambiar el tema de la ventana, como en la ventana de las estructuras.
  - Corregir el test que se ha realizado. La corrección se muestra sobre el mismo panel, indicando cuales han sido las respuestas correctas y cuáles no, en este caso se marcará la respuesta correcta (ver **figura 116**).

Figura 116 – Ventana que muestra un test corregido.



- Solucionar un test abierto por el usuario, marcando las respuestas correctas de cada pregunta (ver **figura 117**).

Figura 117 – Ventana que muestra las soluciones del test.



- Reiniciar el test corregido o solucionado.

3.4.7.- Funcionalidad Adicional

Además de la funcionalidad de los menús y de la barra de herramientas, **VEDYA** nos permite:

3.4.7.1.- Controles de Tamaño y Posición

Variar tanto el tamaño como la posición de algunas representaciones de estructuras de datos. Dentro del panel animación, se han añadido unos controles, con los que podemos desplazar y aumentar o disminuir las estructuras de datos.

En el caso de las representaciones gráficas de las pilas y las colas están diseñadas de tal forma que en una pantalla normal se puedan visualizar correctamente todas las animaciones. Esto es debido a que se ha limitado el número de elementos que pueden almacenar y que estos se colocan de una forma adecuada por todo el panel de animaciones. Por esta razón, en estas representaciones no se facilita al usuario esta funcionalidad.

En estructuras más complejas, como las arbóreas, donde para una perfecta comprensión no hay límite en el número de elementos que contienen, es necesario esta funcionalidad. Desde la **figura 119** hasta la **figura 126**, se muestra el efecto de aplicar estos controles sobre el árbol binario de búsqueda que se ha creado en la **figura 118.**

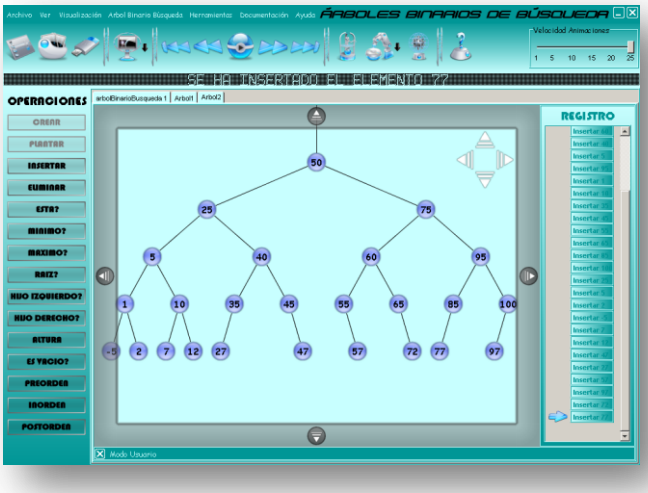


Figura 118 – Árbol sobre el que se van a utilizar los controles de tamaño y posición.

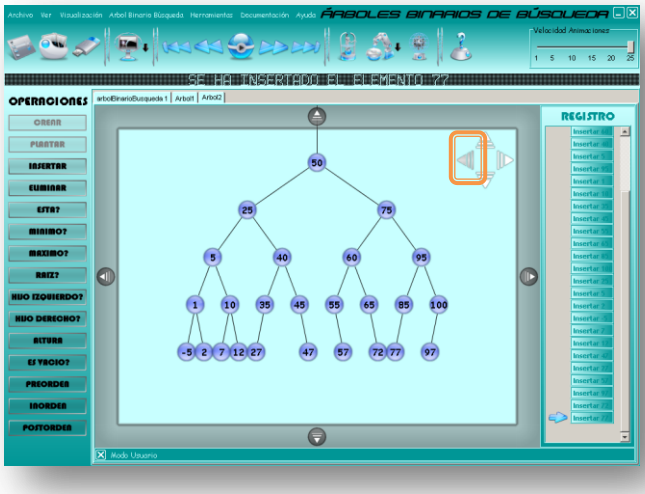


Figura 1199 – Disminución horizontal del árbol de la figura 118.

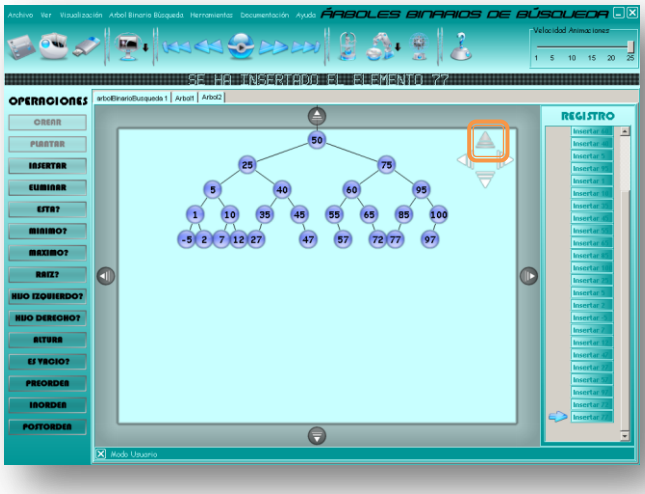


Figura 120 – Disminución vertical del árbol de la figura 118.

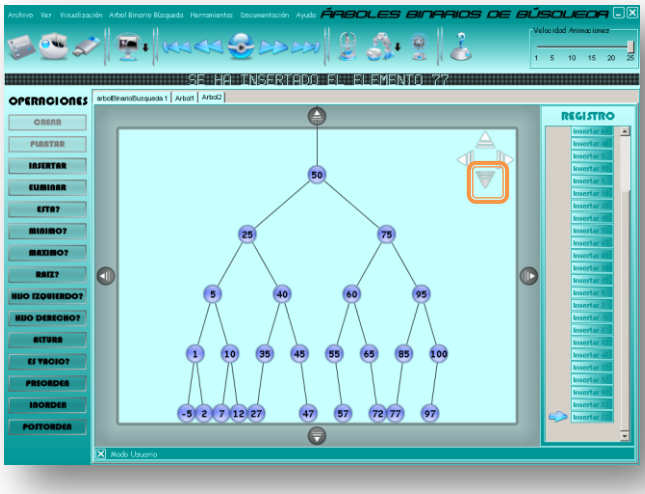


Figura 121 – Alargamiento del árbol de la figura 118.

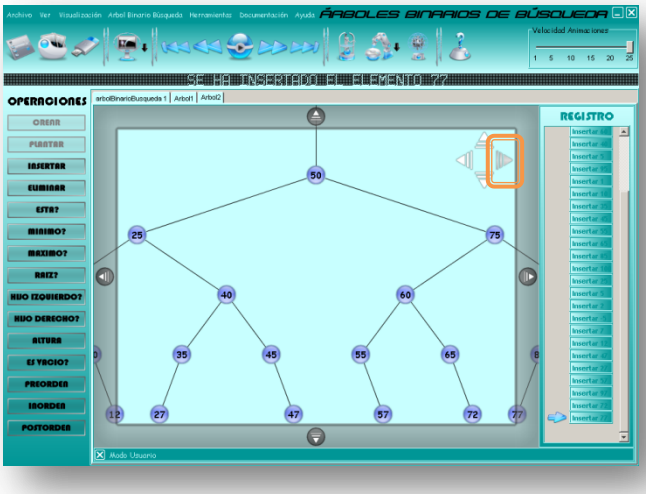


Figura 122 – Ensanchamiento del árbol de la figura 118.

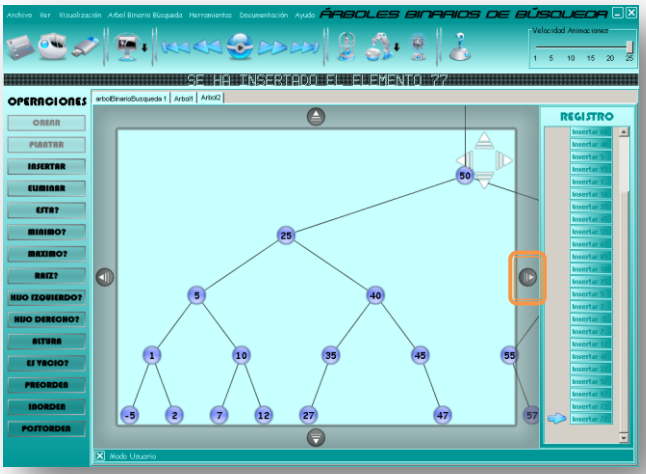


Figura 123 – Desplazamiento hacia la derecha del árbol de la figura 118.

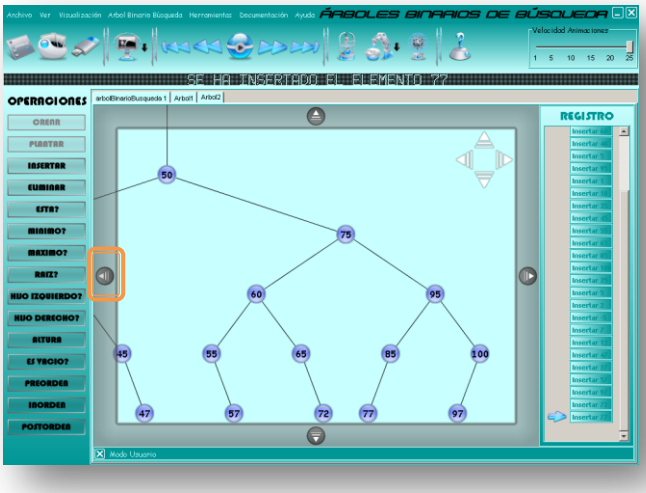


Figura 124 – Desplazamiento hacia la izquierda del árbol de la figura 118.

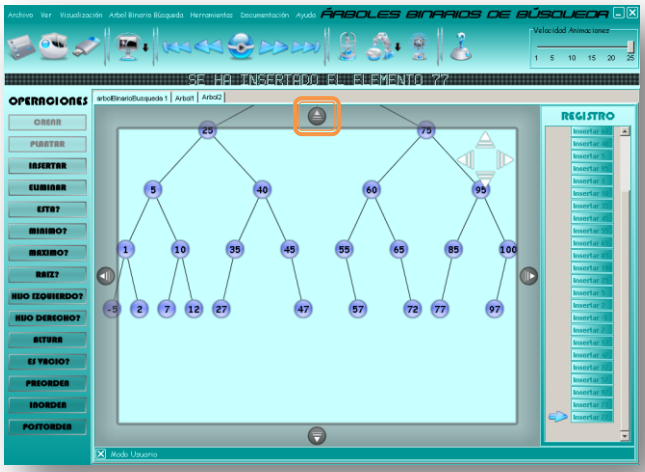


Figura 125 – *Desplazamiento hacia arriba del árbol de la figura 118.*

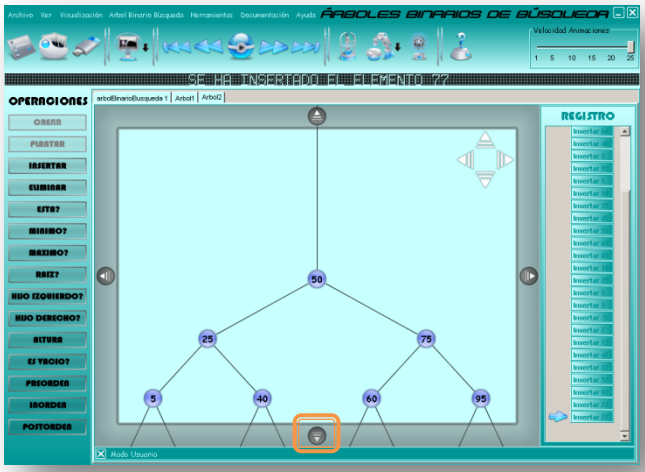


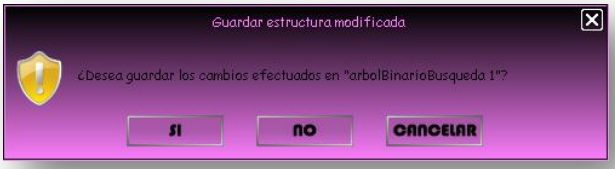
Figura 126 – *Desplazamiento hacia abajo del árbol de la figura 118.*

3.4.7.2.- Control de Archivos

Para que el usuario no pierda información que le pueda ser necesaria, se ha creado este sistema. Se encarga de controlar las simulaciones que no se han guardado en diferentes situaciones

- *Cuando se cierra una pestaña, aparece una ventana como la que se muestra en la* ***figura 127****, en la que el usuario debe elegir debe elegir si desea guardar la simulación, si no o si quiere cancelar el cierre de la pestaña. En caso afirmativo, si la pestaña no tiene asociado un archivo, se produce una secuencia semejante a la de guardar como. En caso contrario, se guarda en el archivo asociado.*

Figura 127 – *Ventana de confirmación para guardar una simulación al cerrar una pestaña.*

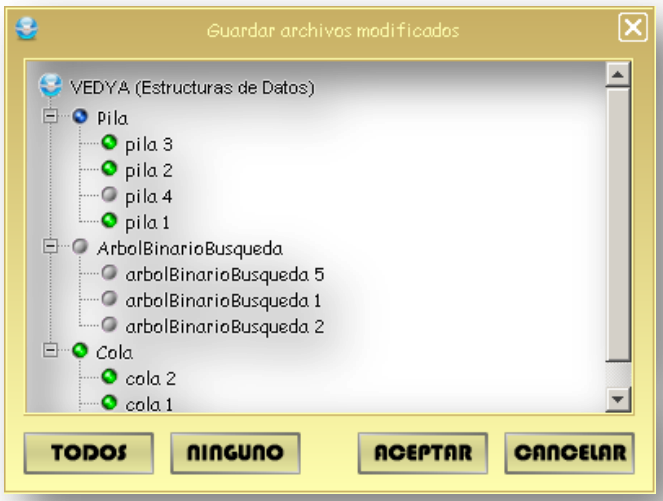


- *Cuando se cierra una ventana de estructura de datos, aparece una ventana como la que se muestra en la* ***figura* 128**, *en la que se listan las simulaciones que no se han guardado. Si el usuario acepta se guardaran las simulaciones seleccionadas (icono verde) y se perderán las no seleccionadas (icono gris).*

Figura 128 – *Ventana de confirmación para guardar varias simulaciones al cerrar una ventana.*

- *Cuando se cierra la aplicación, aparece una ventana como la que se muestra en la* ***figura* 129**, *en la que se listan las simulaciones, ordenadas por estructuras, que no se han guardado. Si el usuario acepta se guardaran las simulaciones seleccionadas (icono verde) y se perderán las no seleccionadas (icono gris).*

Figura 129 – Ventana de confirmación para guardar varias simulaciones al cerrar la aplicación.



3.5.- Funcionalidad de VEDYA-Test

Esta aplicación es un complemento de **VEDYA**, destinada a los profesores, para poder crear test que realizarán los alumnos con el fin de autoevaluarse.

Para una mayor comodidad para crear test, existe una base de datos donde se almacenan todas las preguntas que haya creado el usuario. Esta base de datos está organizada por grupos para facilitar la localización de las preguntas.

Esta aplicación, está distribuida en distintas ventanas con una funcionalidad específica que desarrollamos a continuación.

3.5.1.- Ventana Test

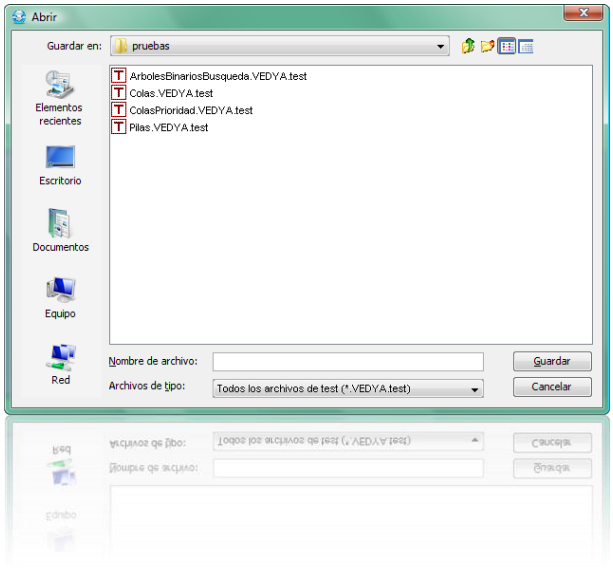
En ésta ventana el usuario podrá crear los test. Estos se visualizan igual que los vería el alumno. A continuación, se detalla el funcionamiento que ofrece esta ventana.

Posee un menú, con las siguientes opciones:

- **Archivo**: menú que contiene todas las opciones para la gestión de los archivos que contienen los test.
  - **Nuevo**: crea un nuevo test en el cual se pueden añadir las preguntas.

- **Abrir**: abre una ventana para elegir el test que se desea cargar. Para una mayor claridad, en esta ventana solo se muestran archivos de test (tienen la extensión asociada "***.VEDYA.Test**"). Una vez seleccionado el test, lo carga y lo muestra (ver **figura 130**).

Figura 130 – Ventana de abrir un test.

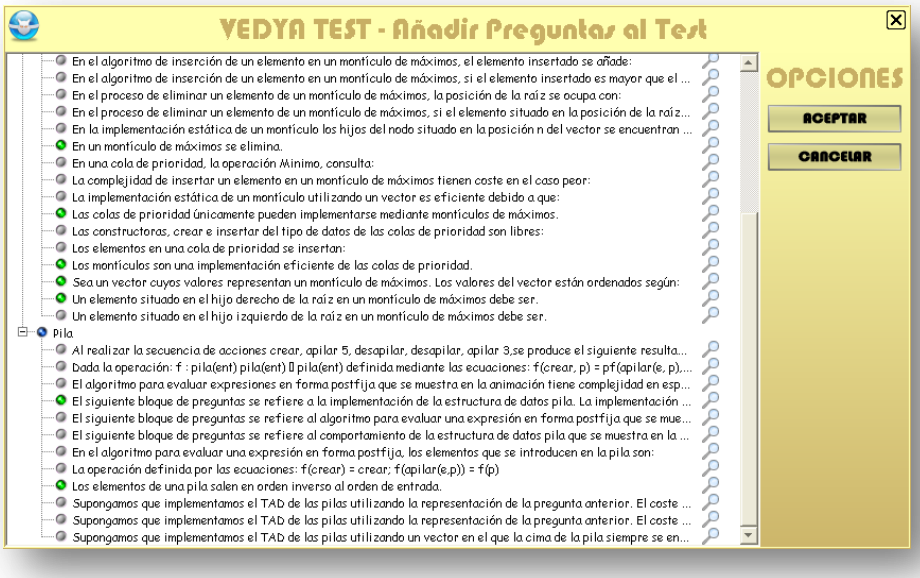


- **Guardar**: guarda el test en el archivo que se guardó la última vez o del cual se cargó, si no es así se solicitará con que nombre se guardará. El test se guardará con la extensión propia de los test.
- **Guardar como**: se pedirá al profesor el nombre del archivo donde se desea guardar el test.
- **BBDD Preguntas**: abre una ventana que contiene las preguntas de la base de datos.
- **Salir**: cierra la aplicación, comprobando si se ha guardado el test. En caso de no haberse guardado, se pide al usuario si desea salvarlo o cerrar la aplicación perdiendo la información.

- **Test**: menú que contiene todas las opciones para modificar los test.
  - **Añadir**: muestra en otra ventana el comienzo de los enunciados de las preguntas de la base de datos, organizados por los grupos a los que pertenecen (ver **figura 131**). Se da la posibilidad de ver cada pregunta por completo si pinchamos sobre el botón que tiene asociado (una lupa). Es en esta ventana donde se puede seleccionar las preguntas que

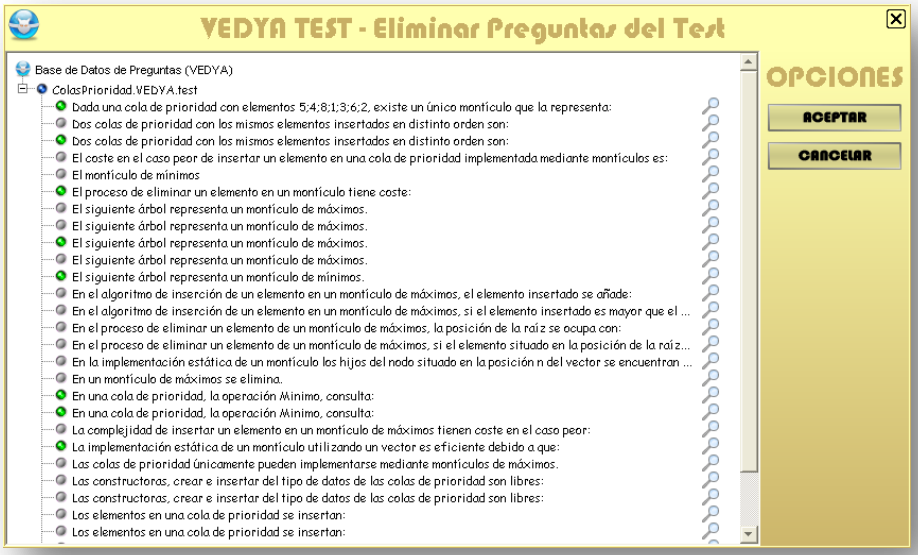
queremos añadir al test. Se añadirán todas aquellas preguntas marcadas en verde y que no se hayan añadido al test desde la última vez que se cargó.

Figura 131 – Ventana de selección de las preguntas para añadir al test.



- ***Eliminar***: muestra en otra ventana el comienzo de los enunciados de las preguntas del test (ver **figura 132**). Se da la posibilidad de ver cada pregunta por completo si pinchamos sobre el botón que tiene asociado (una lupa). Es en esta ventana donde se puede seleccionar las preguntas que queremos eliminar del test. Se eliminarán todas aquellas preguntas marcadas en verde.

Figura 132 – *Ventana de selección de las preguntas para eliminarlas del test.*



- ***Corregir***: *corrige el test realizado, indicando cuales han sido las respuestas correctas y cuáles no, en este caso se marcará la respuesta correcta. (Ver* ***figura 133****)*

Figura 133 – *Ventana con un test corregido.*

- **Solución**: *muestra el test solucionado. (Ver* ***figura*** **134)**

Figura **134** – *Ventana con un test solucionado.*

- **Reiniciar**: *reinicia el test corregido o solucionado. (Ver* ***figura*** **135)**

Figura **135** – *Ventana con un test reiniciado.*

- **Temas**: *nos permite cambiar los colores de la interfaz de usuario.*

3.5.2.- Ventana Base de Datos de Preguntas

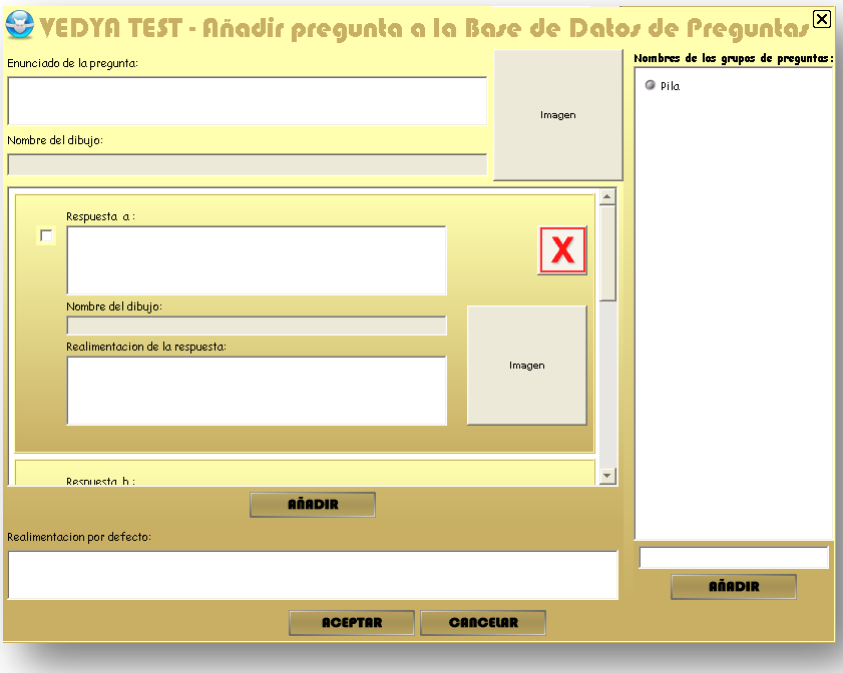
En ésta ventana se muestran todas las preguntas que posee la base de datos de preguntas, dando la posibilidad de modificar su contenido o de eliminarlas. (Ver **figura 136**)

Figura 136 – Ventana de la base de datos de preguntas.

Posee un menú, con las siguientes opciones:

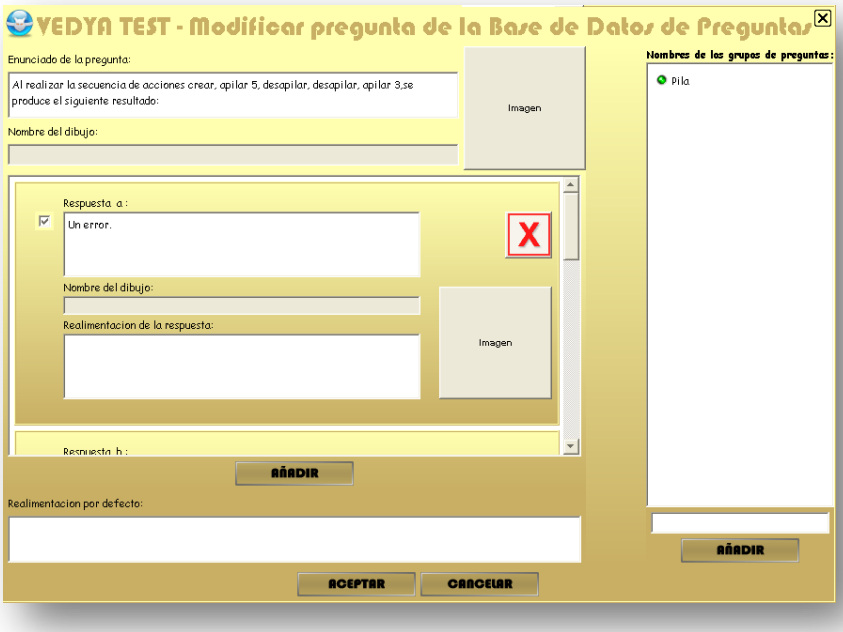
- **Archivo**: menú que contiene opción volver a Test, que vuelve a la Ventana Test, cerrando ésta.
- **Preguntas**: menú que contiene todas las opciones para modificar la base de datos de preguntas.
  - **Añadir**: muestra una ventana para crear una pregunta (ver **figura 137**). Se da la libertad de elegir al usuario el número de respuestas que tenga cada pregunta, así como la posibilidad de introducir imágenes tanto en el enunciado como en las distintas respuestas. Para que la pregunta se pueda guardar, los campos del enunciado y de las respuestas deben estar rellenos, haber seleccionado la respuesta correcta y el grupo al que pertenece.

Figura 137 – *Ventana que permite crear una pregunta.*



- ***Modificar***: *muestra una ventana, por cada pregunta marcada, semejante a la que se abre al añadir, donde los campos están rellenos con los datos de la pregunta a modificar (ver* ***figura 138****).*

Figura 1388 – *Ventana que permite crear una pregunta.*



- **Eliminar**: se eliminarán de la base de datos todas las preguntas marcadas en verde.

- **Temas**: nos permite cambiar los colores de la interfaz de usuario.

3.5.3.- VENTANA VER PREGUNTA

Como ya se ha comentado anteriormente cuando desde alguna ventana se listan las preguntas solo se muestra parte del enunciado de estas. Esta es la ventana que se abre cuando se pulsa sobre una de las lupas que tienen asociadas las preguntas

Muestra el contenido de la pregunta, enunciado, posibles respuestas e imágenes, incluyendo la realimentación de cada respuesta (explicación o motivo por el cual la respuesta es cierta o falsa) y la general. Además, marca la respuesta correcta. En las **figuras 139**, **140** y **141** se muestran distintos ejemplos de preguntas.

Figura 139 – Ventana para visualizar una pregunta sin imágenes.

Figura 140 – Ventana para visualizar una pregunta con una imagen en el enunciado.

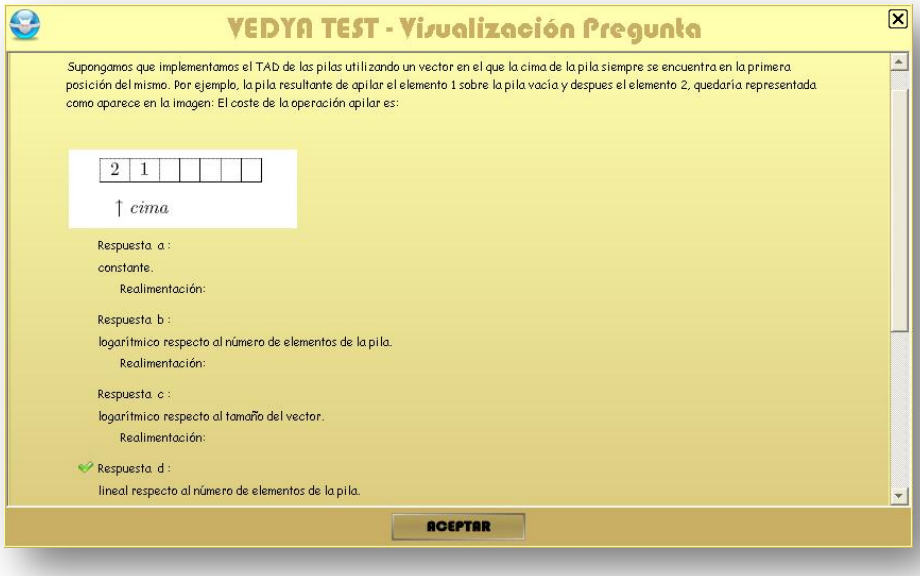
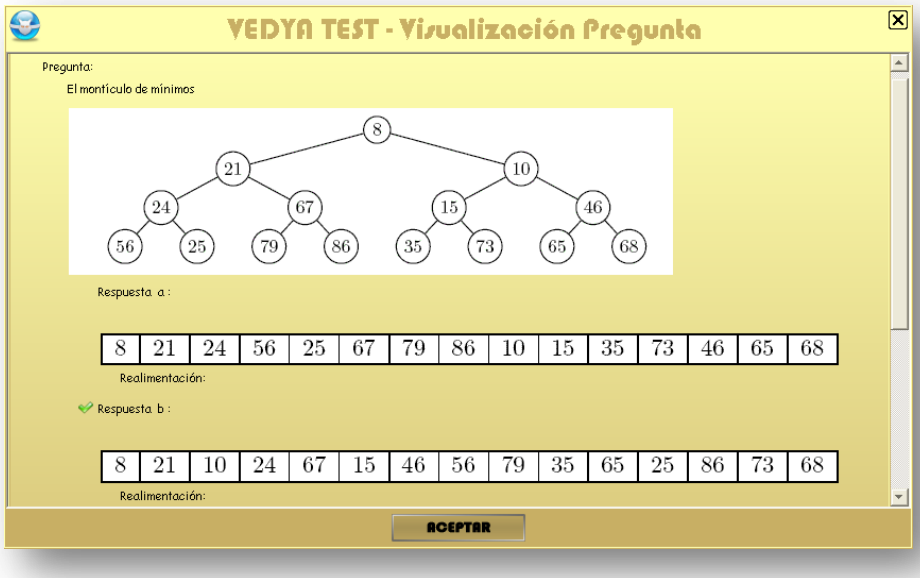


Figura 141 – Ventana para visualizar una pregunta con imágenes en las respuestas.



4.- Estructura de la Aplicación VEDYA

4.1.- Diagrama de Clases

Como hemos comentado anteriormente, para poder plasmar todas las ideas que teníamos para mejorar **VEDYA**, hemos tenido que modificar ligeramente el diseño de la aplicación. Este cambio de diseño ha llevado a un cambio profundo en la forma de representar gráficamente cada una de las estructuras, por lo que el código existente no nos servía. Esto ha supuesto que hemos tenido que reimplementar desde cero prácticamente toda la aplicación, exceptuando las implementaciones de las estructuras.

Debido a este cambio tan radical de la aplicación, la estructura de clases ha cambiado considerablemente.

En la **figura 142** mostramos el diagrama de paquetes de la aplicación, dónde podemos encontrar distribuidas todas las clases de forma que se mantiene la visión del diseño de la aplicación.

En la **figura 143**, se muestra el diagrama de clases de **VEDYA**. Cabe destacar el sentido jerárquico del diseño. Además, si observamos el diagrama con atención, nos damos cuenta que se puede ver bajo dos puntos de vista:

- **Visión vertical:** de esta forma observamos totalmente diferenciados los bloques básicos de la aplicación. Las **ventanas (Rojo)** sirven de canal de comunicación entre la **implementacion (Morado)** y los **graficos (Amarillo)**. Los **hilos (Verde)**, únicamente se relacionan con los **graficos**. Es importante darse cuenta de todas las herencias existentes, pues hemos intentado generalizar lo más posible.
- **Visión horizontal:** si nos fijamos en el diagrama de esta manera nos percatamos de las estructuras de datos son totalmente independientes unas de otras, pudiendo ampliar la aplicación sin tener que tocar el diseño.

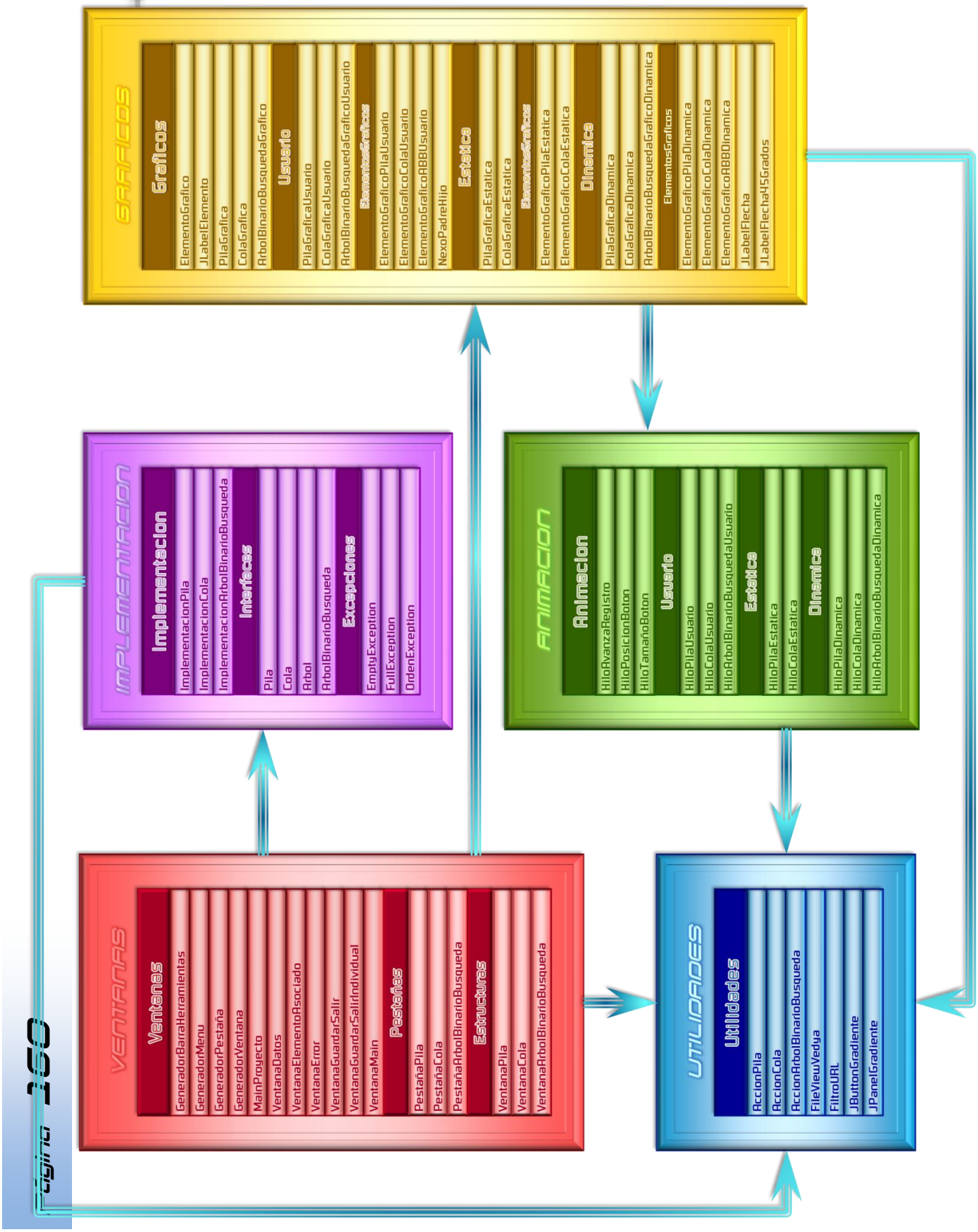
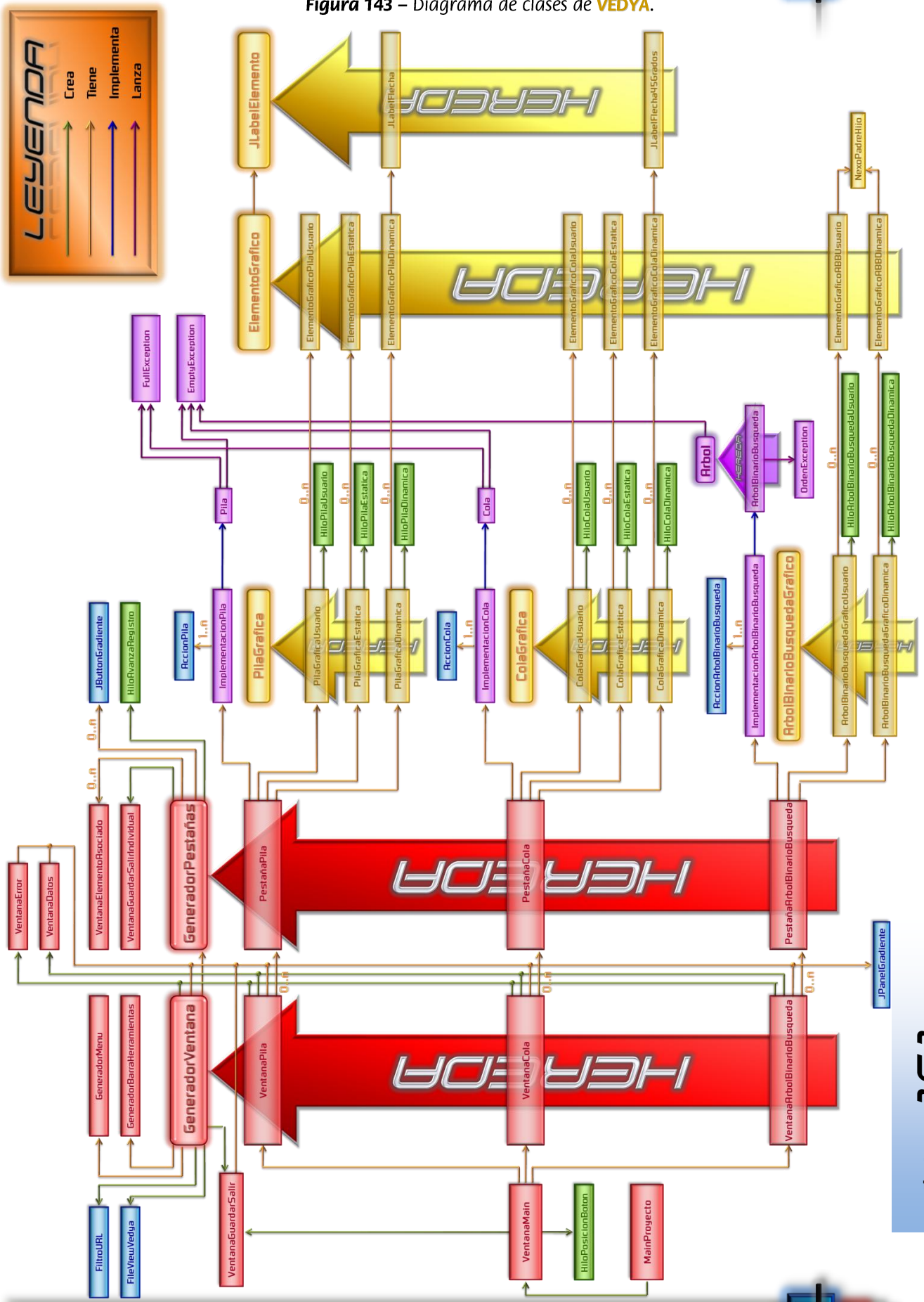


Figura 142 – *Diagrama de paquetes de* **VEDYA**.

Figura 143 – *Diagrama de clases de* **VEDYA**.



4.2.- Modularización y Reutilización del Código

Uno de los mayores problemas que se tuvieron en la realización de la primera versión de **VEDYA**, durante el curso 2004-2005, fue la enormidad de algunas clases, llegando en algún caso a ocupar más de 8000 líneas de código. En la ampliación del proyecto realizada a lo largo del curso 2005-2006, dedicaron mucho tiempo a la modularización de estas gigantescas clases, llegando a reducirlas a unas 2600 líneas. Al comienzo del desarrollo del proyecto que nos ocupa, valoramos muy positivamente esta modularización, pero consideramos que no era suficiente. No lo era porque en cada una de las interfaces gráficas de las estructuras había muchísimo código repetido (nos referimos a configuración y colocación de elementos gráficos como botones, menús, etc…).

Al cambiar el diseño visual de la herramienta, nos hemos preocupado de que la interfaz gráfica de todas las estructuras fuese la misma, de manera que esto nos ha permitido, no solo conseguir modularizarlo mucho más, sino también poder utilizar propiedades de la Programación Orientada a Objetos, como la herencia. De este modo hemos creado la clase **GeneradorVentana** de la cual heredan cada una de las clases que implementan la interfaz gráfica de cada una de las estructuras de datos. La clase **GeneradorVentana** construye la parte de la interfaz que es común a todas las estructuras de datos, mientras que el resto de las ventanas que implementan las interfaces, se centran exclusivamente en lo propio de cada estructura.

En un principio, en **GeneradorVentana** se realizaba la construcción de todos los elementos gráficos comunes que constituyen el interfaz de las estructuras de datos, resultando una clase demasiado grande (en torno a las 4000 líneas). Esto nos llevó a tomar la decisión de continuar modularizando un poco más, de modo que se crearon dos clases nuevas, **GeneradorMenu** y **GeneradorBarraHerramientas**. Estas clases se ocupan de configurar y construir tanto el menú como la barra de herramientas. **GeneradorMenu** se ha quedado finalmente con 1800 líneas mientras que **GeneradorBarraHerramientas** se ha quedado con 800. De esta forma se ha visto reducido drásticamente el número de líneas de la clase **GeneradorVentana** a 1100.

De forma totalmente independiente a las ventanas, hemos implementado todo lo referente a las pestañas que éstas engloban. Así, al igual que tenemos una clase concreta para la interfaz de cada una de las estructuras de datos, vamos a tener una clase concreta para cada una de las pestañas que contienen cierto tipo de estructuras de datos. Como nos ha ocurrido con las ventanas, se ha decidido volver a utilizar la herencia para permitir una mayor modularidad con las pestañas. Se ha creado la clase abstracta **GeneradorPestaña** que tiene todos los atributos y funciones que son comunes a todas las pestañas, independientemente de la estructura de datos con la que estemos tratando. Lo propio de cada una de las estructura de datos, está contenido en las clases destinadas a ello, que heredan de **GeneradorPestaña**.

Para entender el funcionamiento de las ventanas y pestañas, hay que tener claro el concepto de polimorfismo. Hemos considerado que las ventanas contienen pestañas, y esto es un hecho general, por lo que lo hemos llevado a la práctica. La clase **GeneradorVentana** tiene un **JTabbedPane** que es el contenedor (funciona como un vector) de **GeneradorPestaña**. Esto es así independientemente de la estructura de datos con la que vayamos a trabajar. Solamente cuando el usuario ha pulsado el botón de abrir una determinada estructura de datos, se crea la ventana de esa estructura con las pestañas particulares a dicha estructura.

En este afán de descuartizar el código en trozos lo más pequeños posibles (tanto para un mejor entendimiento del mismo para posibles ampliaciones, como para que cada módulo sea lo más independiente posible del resto de la aplicación), hemos considerado que la mejor forma de tratar cada una de las estructuras de datos sea por separado y no solo las ventanas y las pestañas lo hacen de esta manera, sino absolutamente todas las partes importantes del código tratan de manera independiente cada una de las estructuras de datos. Esto es, para cada estructura de datos hay una parte de la herramienta especializada en el tratamiento de sus gráficos y otra parte especializada en el tratamiento de sus animaciones.

Otro cambio bastante importante que hemos introducido es la forma en la que tratamos a cada una de las posibles visualizaciones que tiene una estructura de datos. Las visualizaciones de una estructura de datos se tratan de forma parecida a como se tratan las ventanas, luego hemos aplicado el mismo concepto de herencia y polimorfismo aplicado en las ventanas para optimizar el código de la parte gráfica.

Como resultado, cada estructura de datos tiene la clase **EDGrafica**, que es abstracta, de la cual van a heredar todas las visualizaciones existentes para esa estructura. Por ejemplo, las Pilas tienen la clase **PilaGrafica**, de la que heredan **PilaGraficaUsuario**, **PilaGraficaEstatica** y **PilaGraficaDinamica**.

En las versiones anteriores de **VEDYA**, para cada una de las distintas visualizaciones posibles de una estructura, había un duplicado de los elementos que integraban las ventanas. Esto es, para cada visualización existía un **JPanel** de dibujo donde se mostraba la estructura de datos, un **JPanel** donde se mostraba la estructura en el estado anterior, y los **JButton's** y **JRadioButton's** necesarios para seleccionar la operación que queríamos realizar con la estructura. Cada vez que se realizaba una operación, se ejecutaban las acciones para las distintas visualizaciones y se mostraban los resultados sobre cada elemento. Esta visión, facilitaba el cambiar de una visualización a otra, ya que no había que hacer otra cosa más que añadir los paneles y los elementos correspondientes a la ventana. Por el contrario, suponía tener muchos elementos duplicados, y no solo los paneles, sino también cada uno de los elementos gráficos utilizados para mostrar la estructura dibujados dentro, con la consecuente utilización de memoria. Además, cada una de las operaciones se tenía que realizar para cada una de las visualizaciones, provocando una cierta ineficiencia en la ejecución de la operación.

Hemos creído que estos inconvenientes no son necesarios, y que por renunciar a ellos el cambiar de visualización no tiene porque ser una implementación costosa y complicada.

Así, en esta última implementación, los elementos utilizados para visualizar la estructura en sus diferentes visualizaciones no están duplicados. La pestaña de la estructura de datos será la que tenga el panel donde se dibuja la estructura, además contendrá un atributo con un objeto de la clase **EDGrafica**. Este objeto, gracias al polimorfismo, dependiendo de la visualización en la que nos encontremos sabrá si es **EDGraficaUsuario**, **EDGraficaEstatica** o **EDGraficaDinamica** y sabrá como dibujar la estructura. De ésta forma no hay que realizar las operaciones para todas las visualizaciones. Y para cambiar de visualización sigue siendo igual de sencillo: sólo tendremos que cambiar el objeto de la clase **EDGrafica** y dibujar la estructura.

4.3.- Paquetes y Clases

Para tener una visión global del funcionamiento interno de la aplicación, a continuación haremos una breve descripción de cada clase.

4.3.1.- Paquete Ventanas

El paquete ventanas contiene todas las clases y subpaquetes que hacen referencia al interfaz grafico de usuario. En él, se encuentra toda la funcionalidad interactiva de la aplicación.

4.3.1.1.- VentanaMain

Esta clase, que hereda de **JFrame**, encierra todos los elementos, en gran medida **JButton**, para comenzar la parte didáctica de la herramienta. Es la que brinda la oportunidad al usuario de elegir las estructuras de datos que quiere explorar.

Esta clase es la que se encarga de gestionar qué ventanas y estructuras de datos, han sido abiertas por el usuario.

Para su propósito es necesario que contenga como atributos estáticos, las ventanas de cada una de las estructuras, siendo inicializadas cuando el usuario las seleccione. Además necesita de la clase **VentanaGuardarSalir**, para que al salir de la aplicación, las estructuras que ha creado el usuario puedan ser salvadas. Para conseguir un aspecto más dinámico, se hace uso de la funcionalidad del **HiloPosicionBoton** que hace que los botones entren o salgan de la ventana,

4.3.1.2.- GeneradorMenu

Esta clase, que hereda de **JMenuBar**, contiene, los elementos que forman parte de los menús de las Ventanas de las estructuras de datos.

Es la que se encarga de colocar y organizar estos menús.

4.3.1.3.- GeneradorBarraHerramientas

Esta clase, que hereda de **JPanel**, contiene los **JButton** que forman parte de la barra de herramientas de las Ventanas de las estructuras de datos, así como el control de velocidad de las animaciones.

Esta clase es la que se encarga de colocar y organizar estos elementos.

4.3.1.4.- GeneradorVentana

Esta clase abstracta, que hereda de **JFrame**, es la que se encarga de construir e implementar la funcionalidad de las partes comunes a todas las ventanas de las estructuras de datos. Más concretamente se encarga de colocar en su sitio todos los elementos de una ventana de una estructura de datos.

Para ello contiene los elementos básicos que forman una ventana:

- Un menú de tipo **GeneradorMenu**.
- Los **JButton** para cerrar y minimizar la ventana.
- Una barra de herramientas de tipo **GeneradorBarraHerramientas**.
- El cuadro de dialogo, para mostrar las informaciones de las operaciones.
- El panel con las operaciones de la estructura de datos.
- Un contenedor de **GeneradorPestaña**, que será un **JTabbedPane**, para tener la posibilidad de abrir múltiples pestañas.

Además se encarga de dar funcionalidad a todas las opciones que ofrecen los menús y la barra de herramientas, excepto las que son específicas de cada una de las estructuras de datos. Entre las opciones comunes a todas las ventanas están, por ejemplo, lanzar la ventana para abrir una estructura guardada, ocultar los botones de la barra de herramientas, ocultar los elementos de la ventana, ejecutar y controlar una simulación, etc.

4.3.1.5.- GeneradorPestaña

Esta clase abstracta, que hereda de **JPanel**, es la que se encarga de colocar todos los elementos, que forman la pestaña de una estructura de datos, en su sitio. Además, se encarga de implementar la funcionalidad que no es específica a cada

estructura. Así, por ejemplo, **GeneradorPestaña** se encarga de avanzar la flecha que indica hasta qué operación se está visualizando, cerrar la pestaña, guardar la estructura en un fichero, etc.

Para ello contiene los elementos básicos que forman la pestaña:

- Un panel donde se registran las operaciones realizadas.
- Una barra de estado que contiene el botón para cerrar la pestaña, una etiqueta que indica la visualización de la estructura y los botones que abren los elementos asociados a la operación actual.
- Un **panelAnimacion** de tipo **JPanel** donde se visualiza la estructura y que contiene:
  - Controles para el desplazamiento de la estructura.
  - Controles para la modificación del tamaño de la estructura.
  - Los elementos gráficos que forman la visualización de la estructura.

Para añadir a la pestaña la funcionalidad de mostrar los elementos asociados a una operación, **GeneradorPestaña** contiene una **Hashtable** donde la clave indica la posición de la operación que tiene los elementos asociados y el valor es un Vector de ventanas de dichos elementos.

4.3.1.6.- VentanaDatos

Esta clase, que hereda de **JFrame**, es la que se encarga de recoger los datos necesarios para la operación seleccionada en la ventana de la estructura de datos.

4.3.1.7.- VentanaElementoAsociado

Esta clase, que hereda de **JFrame**, contiene el elemento asociado a una operación.

4.3.1.8.- VentanaError

Esta clase, que hereda de **JFrame**, se encarga de mostrar, siguiendo la estética de nuestra aplicación, el mensaje de error que se haya producido, indicando los parámetros que han producido dicho error.

4.3.1.9.- VentanaGuardarSalir

Esta clase, que hereda de **JFrame**, se encarga de mostrar y construir un árbol con las estructuras que no se han salvado. Una vez que el usuario ha seleccionado lo que desea guardar, se encarga de gestionarlo.

4.3.1.10.- VentanaGuardarSalirIndividual

Esta clase, que hereda de **JFrame** y que se lanza cuando se cierra una pestaña y la estructura no ha sido guardada, se utiliza para consultar al usuario que desea hacer con la estructura de datos, encargándose de gestionar su decisión.

4.3.1.11.- GeneradorMenuTest

Esta clase, que hereda de **JMenuBar**, contiene los elementos de los menús de **VentanaTest**.

4.3.1.12.- VentanaTest

Esta clase, que hereda de **JFrame**, se encarga de mostrar los test. Por ello contiene los siguientes elementos:

- Un menú de tipo **GeneradorMenuTest**.
- Los **JButton** para cerrar y minimizar la ventana.
- Un **JScrollPane** con un panel, **JPanel,** que contiene un **JPanelPregunta** por cada pregunta y en el cual, además, se muestra el test corregido o solucionado.
- Unos botones, **JButton**, que sirven para corregir un test, pedir la solución del test y reiniciar el test una vez corregido o solucionado.

Además, se encarga de dar funcionalidad a todas las opciones que ofrecen los menús, como son abrir los test creados por el profesor.

4.3.1.13.- Subpaquete Estructuras

Para una mayor organización del código, se ha creado el subpaquete estructuras de forma que contenga las clases **VentanaPila**, **VentanaCola** y

VentanaArbolBinarioBusqueda. Heredan de **GeneradorVentana** e implementan sus métodos abstractos, es decir, los métodos que son específicos para cada estructura.

Cada clase de este subpaquete añaden, además, los elementos propios de cada estructura a la ventana, como son los botones de las operaciones.

4.3.1.14.- SUBPAQUETE PESTAÑAS

Este subpaquete contiene, para una mayor organización del código, las clases **PestañaPila**, **PestañaCola** y **PestañaArbolBinarioBusqueda**. Heredan de **GeneradorPestaña** e implementan sus métodos abstractos, es decir, los métodos que son específicos para cada estructura.

Cada clase de este subpaquete contiene la propia estructura de datos, que es la que va a indicar como realizar la animación. Además, contiene las clases que implementan la parte gráfica para cada visualización, esto es **EDGraficaUsuario**, **EDGraficaEstatica** y **EDGraficaDinamica** (que saben de qué manera se tiene que dibujar la estructura).

4.3.2.- PAQUETE IMPLEMENTACIÓN

El paquete implementacion contiene todos los subpaquetes y clases que hacen referencia a la implementación de las estructuras de datos y sus operaciones.

4.3.2.1.- SUBPAQUETE INTERFACES

Este paquete contiene todas las interfaces de las estructuras de datos que tenemos, es decir, **Pila**, **Cola**, **Arbol** y **ArbolBinarioBusqueda**.

El motivo de este subpaquete, además de para una mayor organización del código, es por separar lo que es la declaración de las operaciones de cada estructura de datos, de la implementación (las operaciones de una estructura de datos son las que son, pero se pueden implementar de distintas formas).

Hemos decidido crear las interfaces de **Arbol** y **ArbolBinarioBusqueda**, porque aunque en **VEDYA** solo se visualizan árboles binarios de búsqueda, de esta forma hemos podido separar las operaciones propias de los arboles generales (plantar,

hijoIzquierdo, hijoDerecho, esVacio? y los recorridos) de las operaciones de los árboles binarios de búsqueda (insertar, eliminar, está, mínimo y máximo).

4.3.2.2.- Subpaquete Excepciones

Este paquete contiene todas las clases que tratan los errores que se pueden dar en las implementaciones de las estructuras de datos. Así tenemos;

- **EmptyException** hereda de **Exception**. Excepción que se lanza cuando la estructura está vacía y se intenta eliminar o acceder a un elemento. Se utiliza en todas las estructuras.
- **FullException** hereda de **Exception**. Excepción que se lanza cuando la estructura está llena y se intenta insertar un elemento. Sólo se lanzará en las implementaciones estáticas de las estructuras, que son las que tienen capacidad limitada.
- **OrdenException** hereda de **Exception**. Excepción que se lanza cuando se produce un error al intentar construir un árbol binario de búsqueda sin que se cumplan las condiciones de orden. Es decir, cuando el elemento de la raíz no es mayor que el elemento máximo del hijo izquierdo o no es menor que el elemento mínimo del hijo derecho.

4.3.2.3.- ImplementacionPila

Esta clase, que implementa **Pila** y hereda de **AbstractList**, implementa las pilas mediante un array.

Para indicar como se realizan cada una de las operaciones de una pila, tiene un vector de acciones, que va rellenando con las acciones, **AccionPila**, que indican como simular la operación.

Además de implementar las operaciones propias de las pilas, hay que implementar el método `get()` y `size()` de **AbstractList** para poder recorrer los elementos de la pila mediante un iterador.

4.3.2.4.- ImplementacionCola

Esta clase, que implementa **Cola** y hereda de **AbstractList**, implementa las colas mediante un array circular.

Para indicar como se realizan cada una de las operaciones de una cola, tiene un vector de acciones, que va rellenando con las acciones, **AccionCola**, de una operación.

Además de implementar las operaciones propias de las colas, hay que implementar el método `get()` y `size()` de **AbstractList** para poder recorrer los elementos de la cola mediante un iterador.

4.3.2.5.-ImplementacionArbolBinarioBusqueda

Esta clase, que implementa **ArbolBinarioBusqueda** y hereda de **AbstractList**, implementa los árboles binarios de búsqueda mediante punteros. Un árbol binario de búsqueda está compuesto por un elemento raíz, su hijo izquierdo y su hijo derecho que son a su vez árboles binarios de búsqueda.

Para indicar como se realizan cada una de las operaciones de un árbol binario de búsqueda, tiene un vector de acciones, que va rellenando con las acciones, **AccionArbolBinarioBusqueda**, que indican como simular la operación.

Además de implementar las operaciones propias de los árboles binarios de búsqueda, hay que implementar el método `get()` y `size()` de **AbstractList** para poder acceder a los elementos del árbol.

4.3.3.- Paquete Graficos

El paquete graficos contiene todas los subpaquetes y clases que se encargan de dibujar los elementos de cada una de las estructuras.

4.3.3.1.- PilaGrafica

Esta clase abstracta, se encarga de procesar el vector de acciones que le envía la **PestañaPila**.

Las clases **PilaGraficaUsuario**, **PilaGraficaEstatica** y **PilaGraficaDinamica** son las encargadas de implementar sus métodos abstractos. Estos métodos son los que ejecutan las acciones.

4.3.3.2.- ColaGrafica

Esta clase abstracta, se encarga de procesar el vector de acciones que le envía la **PestañaCola**.

Las clases **ColaGraficaUsuario**, **ColaGraficaEstatica** y **ColaGraficaDinamica** son las encargadas de implementar sus métodos abstractos. Estos métodos son los que ejecutan las acciones.

4.3.3.3.- ArbolBinarioBusquedaGrafico

Esta clase abstracta, se encarga de procesar el vector de acciones que le envía la **PestañaArbolBinarioBusqueda**.

Es la clase **ArbolBinarioBusquedaGraficaUsuario**, la que se encarga de implementar sus métodos abstractos. Estos métodos son los que ejecutan las acciones.

4.3.3.4.- ElementoGrafico

Los objetos de esta clase son los que se dibujan para representar un elemento de una estructura de datos.

Esta clase está formada por un **JLabel**, que contiene el texto del elemento, y un **JLabelElemento**, que contiene la imagen del elemento.

4.3.3.5.- JLabelElemento

Esta clase hereda de **JLabel**. Los objetos de esta clase están formados por una imagen y tienen la funcionalidad de los **JLabel**. Adicionalmente se puede cambiar el ángulo de giro y el tamaño de la imagen.

4.3.3.6.- SUBPAQUETE USUARIO

Este subpaquete encierra las clases que se encargan de dibujar las implementaciones de usuario de las estructuras de datos.

Contiene las clases **PilaGraficaUsuario**, **ColaGraficaUsuario** y **ArbolBinarioBusquedaGraficoUsuario**. Cada una de ellas hereda de **PilaGrafica**, **ColaGrafica** y **ArbolBinarioBusquedaGrafico**, respectivamente, e implementan las acciones definidas para cada estructura de datos, de tal forma que muestran la estructura de datos desde el punto de vista de un usuario. Son las clases que realmente saben cómo se dibuja la estructura de datos en este tipo de visualización. Además, es dentro de los métodos que implementan las acciones donde se crea el hilo correspondiente que anima la acción.

4.3.3.6.1- SUBPAQUETE ELEMENTOSGRAFICOS

Dentro del subpaquete usuario, tenemos el subpaquete elementosGraficos, que encierra las clases que implementan los elementos que se utilizan para dibujar las estructuras de datos en la vista de usuario. Así tenemos:

- **ElementoGraficoPilaUsuario** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar las pilas en la vista de usuario. Para ello, además de los atributos y funcionalidad de **ElementoGrafico**, necesita otros atributos, tramo y ángulo, para saber en cada momento la posición relativa del elemento respecto del dibujo de la estructura.
- **ElementoGraficoColaUsuario** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar las colas en la vista de usuario. Para ello, además de los atributos y funcionalidad de **ElementoGrafico**, necesita del atributo tramo.
- **NexoPadreHijo** hereda de **JLabel**. Es el elemento utilizado para dibujar las rectas que unen los padres con los hijos en un árbol.
- **ElementoGraficoABBUsuario** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar los arboles binarios de búsqueda en la vista de usuario. Para ello, además de los atributos y funcionalidad de

ElementoGrafico, necesita un **NexoPadreHijo** para conectar el elemento con su padre.

4.3.3.7.- Subpaquete Estatica

Este subpaquete encierra las clases que se encargan de dibujar las implementaciones estáticas de las estructuras de datos.

Contiene las clases **PilaGraficaEstatica** y **ColaGraficaEstatica**. Cada una de ellas hereda de **PilaGrafica** y **ColaGrafica**, respectivamente, e implementan las acciones definidas para cada estructura de datos, de tal forma que la muestran desde el punto de vista de su implementación estática. Son las clases que realmente saben cómo se dibuja la estructura de datos en este tipo de visualización. Además, es dentro de los métodos que implementan las acciones donde se crea el hilo correspondiente que anima la acción.

4.3.3.7.1- Subpaquete ElementosGraficos

Dentro del subpaquete estática, tenemos este subpaquete, que encierra las clases que implementan los elementos que se utilizan para dibujar las estructuras de datos en la vista de implementación estática. Así tenemos

- **ElementoGraficoPilaEstatica** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar las pilas en la vista de implementación estática.
- **ElementoGraficoColaEstatica** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar las colas en la vista de implementación estática.

Estos elementos no añaden nada nuevo a **ElementoGrafico**, el motivo de que existan es para mantener la estructura de clases que hemos seguido para cada una de las estructuras de datos.

4.3.3.8.- Subpaquete Dinamica

Este subpaquete encierra las clases que se encargan de dibujar las implementaciones dinámicas de las estructuras de datos.

Contiene las clases **PilaGraficaDinamica** y **ColaGraficaDinamica**. Cada una de ellas hereda de **PilaGrafica** y **ColaGrafica**, respectivamente, e implementan las acciones definidas para cada estructura de datos, de tal forma que la muestran desde el punto de vista de su implementación dinámica. Son las clases que realmente saben cómo se dibuja la estructura de datos en este tipo de visualización. Además, es dentro de los métodos que implementan las acciones donde se crea el hilo correspondiente que anima la acción.

4.3.3.8.1- Subpaquete ElementosGraficos

Dentro del subpaquete dinámica, tenemos este subpaquete, que encierra las clases que implementan los elementos que se utilizan para dibujar las estructuras de datos en la vista de implementación dinámica. Así tenemos:

- **JLabelFlecha** es la clase que dibuja el puntero al elemento siguiente de una pila, Para ello tiene los atributos puntaFlecha y paloFlecha que son de tipo **JLabelElemento**. Este tipo de objetos dibujan flechas horizontales o verticales.
- **ElementoGraficoPilaDinamica** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar las pilas en la vista de implementación dinámica. Está formado por la imagen y el texto del elemento, y por el puntero al elemento siguiente (**JLabelFlecha**).
- **JLabelFlecha45Grados** hereda de **JLabel**. Es la clase que dibuja el puntero al elemento siguiente de una cola. Este tipo de objetos dibujan flechas giradas 45 grados con respecto a la horizontal o la vertical.
- **ElementoGraficoColaDinamica** hereda de **ElementoGrafico**. Los objetos de este tipo se utilizan para dibujar las colas en la vista de implementación dinámica. Está formado por la imagen y el texto del elemento, y por el puntero al elemento siguiente (**JLabelFlecha45Grados**).

Estos elementos no añaden nada nuevo a **ElementoGrafico**, el motivo de que existan es para mantener la estructura de clases que hemos seguido para cada una de las estructuras de datos.

4.3.4.- Paquete Animacion

En el paquete animación es donde se encuentran los hilos que, no solo le dan a la aplicación un aspecto dinámico, sino que son imprescindibles para el aspecto didáctico de **VEDYA**.

4.3.4.1.- HiloAvanzaRegistro

Durante la simulación de una operación, es este hilo el que se encarga de mover la flecha del registro de operaciones, para que apunte a la operación actual. El motivo de utilizar un hilo para este propósito es para que su movimiento fuese en sintonía con el de la animación de la operación.

4.3.4.2.- HiloPosicionBoton

Este hilo controla el movimiento de los botones de la ventana principal.

4.3.4.3.- Subpaquete Usuario

En este subpaquete nos encontramos con **HiloPilaUsuario**, **HiloColaUsuario** e **HiloArbolBinarioBusquedaUsuario**. Estos hilos son los responsables de las animaciones de cada una de las acciones que se tienen que realizar para simular una operación en la vista de usuario.

4.3.4.4.- Subpaquete Estatica

En este subpaquete nos encontramos con **HiloPilaEstatica** e **HiloColaEstatica**. Estos hilos son los responsables de las animaciones de cada una de las acciones que se tienen que realizar para simular una operación en la vista de implementación estática, para cada una de las estructuras de datos.

4.3.4.5.- Subpaquete Dinamica

En este subpaquete nos encontramos con **HiloPilaDinamica** e **HiloColaDinamica**. Estos hilos son los responsables de las animaciones de cada una de las acciones que se tienen que realizar para simular una operación en la vista de implementación dinámica, para cada una de las estructuras de datos.

4.3.5.- Paquete Utilidades

En este paquete se han incluido todas aquellas clases que por sus características no tenían un lugar definido en ningún otro paquete, o clases que son utilizadas en varias partes de la aplicación.

4.3.5.1.- AccionPila

En esta clase se han incluido los atributos necesarios para definir las acciones de la Pila, así como los métodos necesarios para gestionar su información. Los objetos de este tipo son creados en la **ImplementacionPila**, y son utilizados en **PilaGraficaUsuario**, **PilaGraficaEstatica** y **PilaGraficaDinamica**.

4.3.5.2.- AccionCola

En esta clase se han incluido los atributos necesarios para definir las acciones de la Cola, así como los métodos necesarios para gestionar su información. Los objetos de este tipo son creados en la **ImplementacionCola**, y son utilizados en **ColaGraficaUsuario**, **ColaGraficaEstatica** y **ColaGraficaDinamica**.

4.3.5.3.- AccionArbolBinarioBusqueda

En esta clase se han incluido los atributos necesarios para definir las acciones de los árboles binarios de búsqueda, así como los métodos necesarios para gestionar su información. Los objetos de este tipo son creados en la **ImplementacionArbolBinarioBusqueda**, y son utilizados en **ArbolBinarioBusquedaGraficoUsuario**.

4.3.5.4.- FiltroUAL

A los ficheros que contienen las operaciones que representan una estructura de datos, se les ha asociado una extensión propia. Como una ventana solo puede contener las estructuras de datos correspondientes, esta clase utiliza estas extensiones para que al abrir o guardar un fichero únicamente muestre los ficheros de este tipo de estructuras de datos.

4.3.5.5.- FileViewVedya

Esta clase se encarga de asociar un icono a cada una de las extensiones asociadas a las estructuras de datos.

4.3.5.6.- JButtonGradiente

Esta clase es un **JButton** modificado, de tal manera que se le aplica un degradado de dos colores sobre la superficie del botón, pudiendo configurar distintas orientaciones del degradado.

4.3.5.7.- JPanelGradiente

Esta clase es un **JPanel** modificado, de tal manera que se le aplica un degradado de dos colores sobre la superficie del panel, pudiendo configurar distintas orientaciones del degradado.

4.3.5.8.- JPanelPregunta

Esta clase es un JPanelGradiente modificado, de tal manera que se muestra una pregunta en él, con diferentes características según los siguientes formatos:

- "Ver": crea el panel con todos los elementos que posee la pregunta, como son el enunciado, dibujo del enunciado, las respuestas con su texto, sus dibujos y la realimentación de cada una, y la realimentación de la pregunta.
- "Corregir": crea el panel con el enunciado y las respuestas, con los respectivos dibujos, que posee la pregunta. Además, se muestra la corrección de la pregunta y, si fuera necesario la realimentación.
- "Solución": crea el panel con el enunciado y las respuestas, con los respectivos dibujos, que posee la pregunta y señalando la respuesta correcta.

4.3.6.- Paquete Test

El paquete test contiene todas las clases que hacen referencia a la implementación del test.

4.3.6.1.- Test

Esta clase, que implementa **java.io.Serializable** para poder cargar de disco un objeto de esta clase, representa un test con sus preguntas y respuestas.

Se ha creado mediante una **LinkedList** y un **String** con el nombre del test. En la lista se suceden las preguntas del test, siendo éstas de la clase **Pregunta**.

4.3.6.2.- Pregunta

Esta clase, que implementa **java.io.Serializable** para que los test se puedan cargar de disco, representa una pregunta de un test.

Está construida mediante los siguientes elementos:

- El nombre del grupo al que pertenece la pregunta, de tipo **String**
- El enunciado, de tipo **String**.
- El nombre del dibujo asociado al enunciado, de tipo **String**.
- La realimentación general o por defecto, de tipo **String**.
- Una **LinkedList** donde se guardan las respuestas, de clase **Respuesta**.

4.3.6.3.- Respuesta

Esta clase, que implementa **java.io.Serializable** para que la clase Pregunta pueda ser "Serializable" y por lo tanto los test se puedan cargar de disco, representa una respuesta de una pregunta.

Está construida mediante los siguientes elementos:

- El texto de la respuesta, de tipo **String**
- El nombre del dibujo asociado a la respuesta, de tipo **String**.
- La realimentación de la respuesta, de tipo **String**.
- Un **boolean** para saber si la respuesta es correcta o no.

5.- Estructura de la Aplicación VEDYA-TEST

5.1.- Diagrama de Clases

VEDYA-TEST es una aplicación diseñada para el uso de los profesores, cuyo funcionamiento es totalmente independiente de **VEDYA**. Dentro de su carácter independiente, **VEDYA** tenía que ser capaz de abrir tests creados por **VEDYA-TEST**. Por este motivo, el único elemento común a las dos aplicaciones es la clase test.

Nuestro principal objetivo es crear los test. Para ello hemos creado una base de datos de preguntas, a partir de la cual se construirán todos los tests. Hemos creído que es la forma más conveniente de estructurar **VEDYA-TEST**, ya que tener la preguntas en una base de datos nos permite crear infinidad de tests con múltiples combinaciones de preguntas.

El diagrama de la **figura 144** muestra los paquetes que tiene la aplicación. Como se puede observar, una gran parte de la aplicación son elementos gráficos, como ventanas y paneles, para hacer interactiva la aplicación y con un fácil manejo.

El diagrama de la **figura 145** muestra como se relacionan las clases de **VEDYA-TEST**.

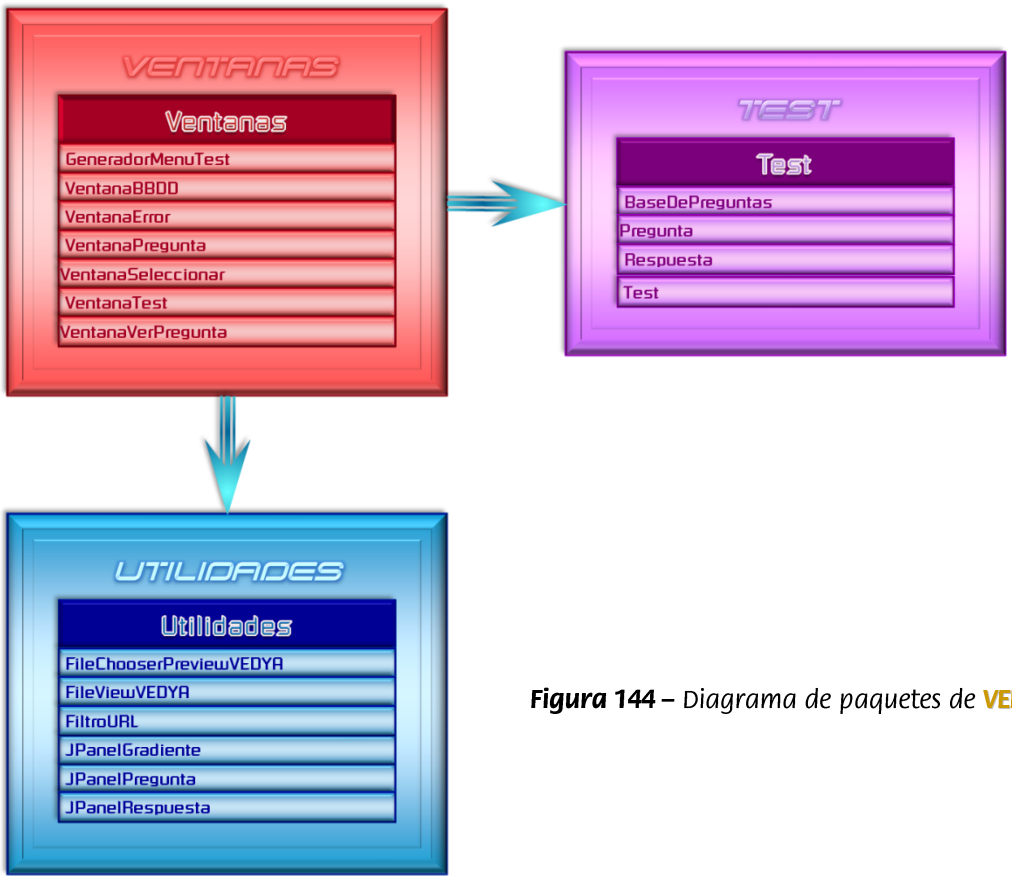


Figura 144 – *Diagrama de paquetes de* **VEDYA-TEST**.

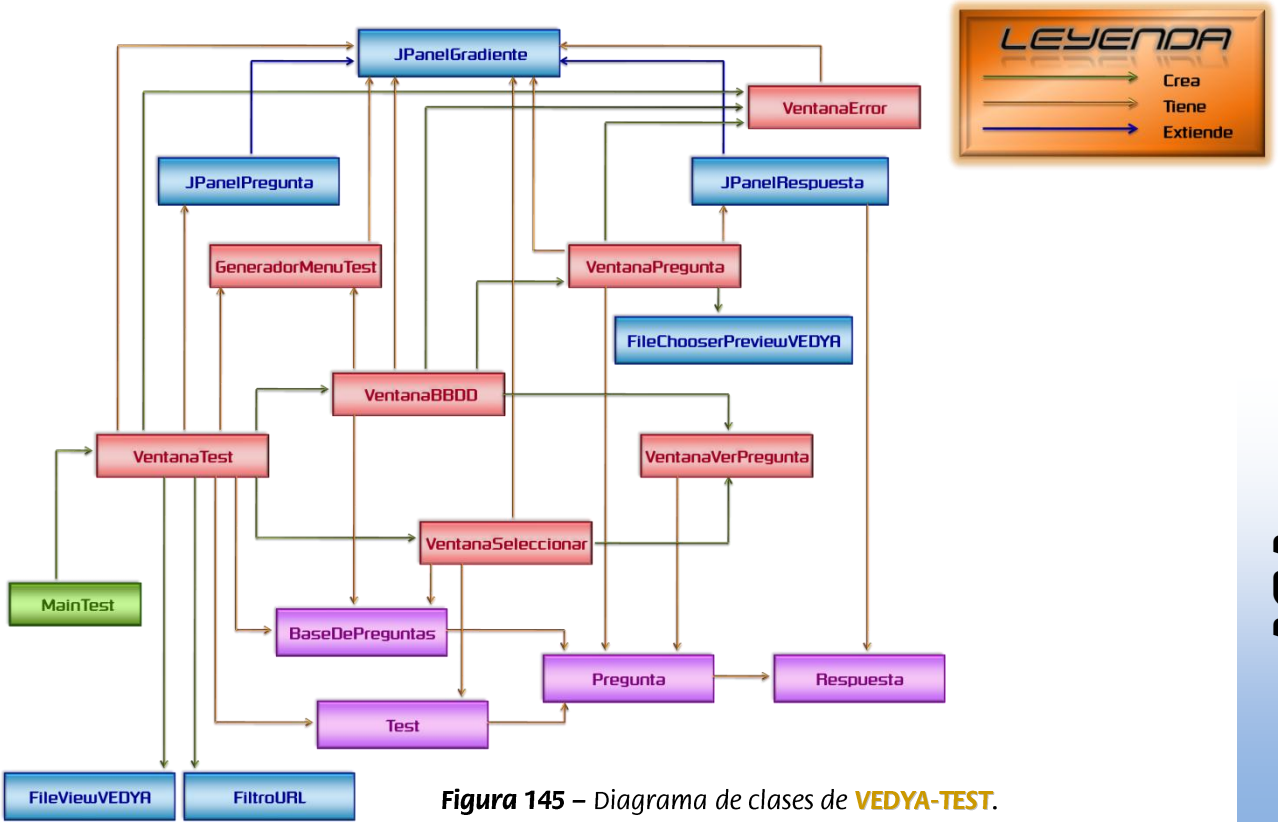


Figura 145 – *Diagrama de clases de* **VEDYA-TEST**.

5.2.- PAQUETES Y CLASES

Para tener una visión global del funcionamiento de la aplicación, a continuación haremos una breve descripción de cada clase.

5.2.1.- PAQUETE VENTANAS

El paquete ventanas contiene todas las clases que hacen referencia al interfaz gráfico de usuario. En él se encuentra la funcionalidad interactiva de la aplicación.

5.2.1.1.- GENERADORMENUTEST

Esta clase, que hereda de **JMenuBar**, contiene los elementos comunes de los menús de las Ventanas **VentanaTest** y **VentanaBBDD**.

Es la que se encarga de colocar y organizar los menús

5.2.1.2.- VENTANATEST

Esta clase, que hereda de **JFrame**, es la que se encarga de crear y modificar los distintos test que realizaran los alumnos. Sirve para saber lo que vería el usuario final, que serán los alumnos.

Por ello contiene los siguientes elementos:

- Un menú de tipo **GeneradorMenuTest**.
- Los **JButton** para cerrar y minimizar la ventana.
- La base de datos de preguntas, **BaseDePreguntas**, que contiene todas las preguntas que se pueden insertar en los test.
- Un **JScrollPane** con un panel, **JPanel,** que contiene un **JPanelPregunta** por cada pregunta y en el cual, además, se muestra el test corregido o solucionado.
- Unos botones, **JButton,** que sirven para añadir o eliminar preguntas del test, para lo cual utiliza la **VentanaSeleccion**, corregir un test como su fuera un alumno, pedir la solución del test, reiniciar el test una vez corregido o solucionado y acceder a la **VentanaBBDD**, para modificar la base de datos de las preguntas.

Además se encarga de dar funcionalidad a todas las opciones que ofrecen los menús, como son abrir los test anteriormente creados, guardar los test nuevos o modificados, etc.

5.2.1.3.- VentanaSeleccionar

Esta clase, que hereda de **JFrame**, se encarga de mostrar en un panel **JPanel** las preguntas de **BaseDePreguntas** o de **Test,** según sea añadir o eliminar al test, respectivamente, en forma de árbol.

En el árbol se muestran los grupos de las preguntas, estos contienen el comienzo del enunciado de las preguntas. Las preguntas tienen asociado un **JButton** que sirve para ver la pregunta individualmente en otra ventana, **VentanaVerPregunta**, estos botones se relacionan con la pregunta mediante una **Hashtable**.

5.2.1.4.- VentanaBBDD

Esta clase, que hereda de **JFrame**, es la que se encarga de gestionar la base de datos de preguntas.

Para ello contiene los siguientes elementos:

- Un menú de tipo **GeneradorMenu**.
- Los **JButton** para cerrar y minimizar la ventana.
- Un panel donde se muestran en forma de árbol todas las preguntas de **BaseDePreguntas** (asociando las hojas de los árboles con las preguntas mediante una **Hashtable**) para poder seleccionarlas. Cada pregunta tiene un **JButton** asociado que sirve para ver la pregunta individualmente en otra ventana, **VentanaVerPregunta**, dichos botones están asociados a las preguntas mediante otra **Hashtable**.
- Los **JButton** para añadir y modificar, mediante **VentanaPregunta,** o eliminar preguntas de la base de datos de preguntas.

Además se encarga de dar funcionalidad a todas las opciones que ofrecen los menús, como son añadir, modificar y eliminar preguntas de la base de datos, guardándola automáticamente al salir.

5.2.1.5.- VentanaPregunta

Esta clase, que hereda de **JFrame**, es la que se encarga de modificar o crear una pregunta.

Para ello contiene los siguientes elementos:

- Un **JScrollPane** que contiene un **JTextPane** donde se inserta el enunciado.
- Un **JButton** se encarga de seleccionar un dibujo para asociarlo al enunciado. Cuando se elige un dibujo, la imagen del botón pasa a ser el dibujo.
- Un **JTextField** donde se muestra el nombre del dibujo asociado al enunciado.
- Un **JScrollPane** con un panel, **JPanel,** que contiene un **JPanelRespuesta** por cada respuesta.
- Un **JButton** que añade otra respuesta más a la pregunta.
- Un **JScrollPane** que contiene un **JTextPane** donde se inserta la realimentación por defecto.
- Un **JScrollPane** con un panel, **JPanel,** contiene un **JRadioButton**, por cada grupo que se tiene en la base de datos de preguntas, para la selección del grupo al que pertenece la pregunta.
- Un **JTextField** donde escribir el nombre de un nuevo grupo y un **JButton** para añadirlo.
- Unos **JButton** para aceptar o cancelar el añadir la pregunta a la base de datos de preguntas.

Para que una pregunta sea correcta y pueda ser añadida, tiene que tener al menos su enunciado, una respuesta correcta y el grupo al que pertenece.

5.2.1.6.- VentanaVerPregunta

Esta clase, que hereda de **JFrame**, es la que se encarga mostrar una única pregunta en una pantalla con todos los atributos que contiene, es decir, el enunciado, dibujos, realimentaciones, respuestas e indicar la correcta.

5.2.1.7.- VentanaError

Esta clase, que hereda de **JFrame**, se encarga de mostrar, siguiendo la estética de nuestra aplicación, el mensaje de error que se haya producido, indicando los parámetros que han producido dicho error.

5.2.2.- Paquete Test

El paquete test contiene todas las clases que hacen referencia a la implementación de la base de datos de preguntas y del test.

5.2.2.1.- BaseDePreguntas

Esta clase, que implementa **java.io.Serializable** para poder guardar y cargar de disco un objeto de esta clase, implementa una base de datos de preguntas.

Está construida mediante una **Hashtable** con los nombres de los grupos como claves y como valor otra **Hashtable.** En esta última, se guardan las preguntas, de la clase **Pregunta**, por el índice de la pregunta, que es distinto en todas ellas. El índice es un atributo interno que el usuario no conoce en ningún momento.

Se pueden añadir o eliminar preguntas de la base de preguntas, pedir todos los grupos que tiene, pedir todas las preguntas de un grupo o una única pregunta.

5.2.2.2.- Test

Esta clase, que implementa **java.io.Serializable** para poder guardar y cargar de disco un objeto de esta clase, representa un test.

Está construida mediante una **LinkedList** y un **String** con el nombre del test. La lista contiene las preguntas, de la clase **Pregunta**, que forman el test.

Esta construido de esta forma para que sea un objeto totalmente independiente de la base de datos.

Se puede añadir y eliminar preguntas del test, cambiar el nombre del test, y pedir la lista de las preguntas, así como el nombre del test.

5.2.2.3.- Pregunta

Esta clase representa una pregunta. Implementa **java.io.Serializable** para que los test y la base de datos de preguntas se puedan guardar y cargar de disco como un objeto de esta clase.

Está construida mediante los siguientes elementos:

- Varios **String** con el nombre del grupo al que pertenece la pregunta, el enunciado, el nombre del dibujo asociado al enunciado y la realimentación general o por defecto.
- Un **Long** con el número de la pregunta para poder almacenarla en la base de datos de preguntas
- Una **LinkedList** donde se guardan las respuestas, de clase **Respuesta**, según el orden de la opción dentro de la pregunta.

Se puede pedir y modificar cada uno de los elementos que tiene.

5.2.2.4.- Respuesta

Esta clase representa una respuesta. Implementa **java.io.Serializable** para que la clase Pregunta pueda ser "Serializable" y por lo tanto los test y la base de datos de preguntas se puedan guardar y cargar en disco como un objeto de esta clase,.

Está construida mediante los siguientes elementos:

- Varios **String** con el texto de la respuesta, el nombre del dibujo asociado a la respuesta y la realimentación de la respuesta.
- Un **boolean** para saber si la respuesta es correcta o no.

Se puede pedir y modificar cada uno de los elementos que tiene.

5.2.3.- Utilidades

5.2.3.1.- FileChooserPreviewVEDYA

Esta clase se encarga de previsualizar los archivos de imágenes a la hora de cargar una imagen.

5.2.3.2.- FileViewVedya

Esta clase se encarga de asociar un icono a los archivos con las extensiones que se le indican, para que a la hora de abrir o guardar un archivo sea más fácil reconocer los distintos tipos de archivo.

5.2.3.3.- FiltroURL

A los ficheros que representan un test se le asocia una extensión propia (*.VEDYA.test). Esta clase utiliza esta extensión para que al abrir o guardar un fichero únicamente muestre los ficheros con esta extensión.

5.2.3.4.- JPanelGradiente

Esta clase es un JPanel modificado, de tal manera que se le aplica un degradado de dos colores sobre la superficie del panel, pudiendo configurar distintas orientaciones del degradado.

5.2.3.5.- JPanelPregunta

Esta clase es un JPanelGradiente modificado, de tal manera que se muestra una pregunta en él, con diferentes características según los siguientes formatos:

- "Ver": crea el panel con todos los elementos que posee la pregunta, como son el enunciado, dibujo del enunciado, las respuestas con su texto, sus dibujos y la realimentación de cada una, y la realimentación de la pregunta.
- "Corregir": crea el panel con el enunciado y las respuestas, con los respectivos dibujos, que posee la pregunta. Además, se muestra la corrección de la pregunta y, si fuera necesario la realimentación.
- "Solución": crea el panel con el enunciado y las respuestas, con los respectivos dibujos, que posee la pregunta y señalando la respuesta correcta.

5.2.3.6.- JPanelRespuesta

Esta clase es un JPanelGradiente modificado, de tal manera que se muestran los elementos que puede tener una respuesta, para modificarla o crear una nueva, contiene los siguientes elementos:

- Un **JScrollPane** que contiene un **JTextPane** donde se inserta el texto de la respuesta.
- Un **JButton** se encarga de seleccionar un dibujo para asociarlo a la respuesta y en el cual se muestra.
- Un **JTextField** donde se muestra el nombre del dibujo asociado a la respuesta.
- Un **JScrollPane** que contiene un **JTextPane** donde se inserta la realimentación de la respuesta.
- Un **JCheckBox** para señalar si la respuesta es correcta o no.

6.- Aspectos Técnicos de la Implementación

6.1.- Subsistema de Carga y Almacenamiento

Una nueva característica que se ha añadido a **VEDYA**, y que creemos que será muy útil, es el subsistema de carga y almacenamiento.

Este subsistema nos da la oportunidad de guardar simulaciones de estructuras de datos, y cargar dichas simulaciones. Además se encarga de llevar el control del almacenamiento de todas las simulaciones de estructuras de datos, abiertas. Este control consiste en que la aplicación sabe de todas las simulaciones abiertas, que simulaciones están guardadas, que simulaciones han sido guardadas pero posteriormente modificadas, y que simulaciones no han sido nunca guardadas.

El almacenamiento de una simulación de una estructura de datos, necesita el tipo de los elementos que contiene y la secuencia de operaciones que se ha realizado. Para guardar esta información se han elegido ficheros de texto planos sin ningún tipo de codificación, de forma que se permite abrir estos archivos con un editor de textos y modificarlos a nuestro antojo.

Para guardar la secuencia de operaciones de una estructura de datos nos servimos del registro de operaciones. Gracias a que en este registro se almacenan todas las operaciones que se han realizado hasta el momento, podemos leer todo el registro, operación a operación, guardando cada una en el fichero. Cada operación ocupará una línea, y en caso de que la operación tenga parámetros, se separarán mediante espacios.

Debido a la posibilidad de que se modifiquen incorrectamente estos archivos, se ha dotado a **VEDYA** de la facultad de comprobar, al abrir un fichero de texto, que es un fichero correcto de **VEDYA**. Para ello, en primer lugar, comprueba que tanto el tipo de los elementos, como la secuencia de operaciones, son correctos, es decir, las

operaciones están correctamente escritas. Y en segundo lugar, se comprueba que la secuencia de operaciones es capaz de construir una estructura de datos correcta sin llegar nunca a casos de error. Para esta última comprobación, se utiliza la implementación de la estructura de datos, aplicándole la secuencia de operaciones.

- Si la estructura de datos es capaz de realizar todas las operaciones sin lanzar una excepción, la secuencia de operaciones es correcta.
- Si la estructura de datos lanza alguna excepción al procesar alguna de las operaciones, capturamos dicha excepción y mostramos un mensaje de error debido a la imposibilidad de abrir el archivo.

Para entender el funcionamiento de estas comprobaciones, veamos en los ejemplos de las **tablas 4, 5 y 6**, cómo serían archivos correctos e incorrectos para cada una de las estructuras implementadas.

Archivo correcto para Pilas	Archivo incorrecto para Pilas
Crear Entero	Crear Entero
Apilar 5	Apilar 5
Apilar 7	Apilar 7
Desapilar	Desapilar
Cima	Desapilar
Apilar 3	Cima
	Apilar 3

Como en la cima de la pila tenemos el 7, se puede desapilar.

Como en la pila tenemos el 5, se puede consultar la cima de la pila.

Como en la cima de la pila tenemos el 7, se puede desapilar.

Como en la cima de la pila tenemos el 5, se puede desapilar.

Como la pila está vacía, al intentar consultar la cima de la pila se lanza una excepción.

Tabla 4 – Archivos de Pilas

Archivo correcto para Colas	Archivo incorrecto para Colas
Crear Entero	Crear Entero
PedirVez 10	PedirVez 10
Primero	Avanzar
Avanzar	Avanzar

Como en la cola tenemos el 10, se puede consultar el primero.

Como en la cola tenemos el 10, se puede avanzar.

Como en la cola tenemos el 10, se puede avanzar.

Como la cola está vacía, al intentar avanzar se lanza una excepción.

Tabla 5 – Archivos de Colas

Archivo correcto para ABB	Archivo incorrecto para ABB
Crear Entero	Crear Entero
Insertar 6	Insertar 6
Eliminar 2	Esta? 2
Eliminar 6	Eliminar 6
	HijoIzquierdo?

En un árbol no es incorrecto eliminar un elemento que no está.

No se puede consultar el HijoIzquierdo de un árbol vacío, salta una excepción.

Tabla 6 – Archivos de Árboles Binarios de Búsqueda

Como hemos comentado, **VEDYA** controla todos los archivos que están abiertos. Este control se utiliza para que, si el usuario sale de **VEDYA** o cierra alguna de sus ventanas o pestañas, sin haber guardado alguna simulación, el sistema advierte al usuario de que se va a perder la información de esas simulaciones. Para ello, cada pestaña cuenta con un atributo `booleano modificado`, que controla si se ha realizado alguna modificación en la secuencia de operaciones desde la última vez que se guardó. Es necesario tener la ruta del archivo para poder acceder a él cuando sea oportuno.

Podemos distinguir tres casos, bien diferenciados, con los que se puede encontrar el subsistema de carga y almacenamiento:

- **Se cierra una pestaña**: este caso es el más sencillo, ya que es la propia pestaña la que se controla a sí misma. Si al cerrarse el valor del atributo modificado es true, se lanza la **VentanaGuardarSalirIndividual**, que pregunta al usuario si se quiere guardar. En caso afirmativo, la ventana guarda la secuencia de operaciones de la pestaña y elimina de su **JTabbedPane** dicha pestaña. En caso negativo únicamente se hace esto último.
- **Se cierra la aplicación**: la ventana principal tiene que recorrer cada una de las ventanas que están abiertas comprobando en cada una de ellas, las pestañas que han sido modificadas. Al hacer este recorrido, se utiliza una **Hashtable** para almacenar toda esta información. Se estructura de la siguiente manera:
  - La ventana principal crea una tabla hash en la que cada clave es el nombre de una estructura de datos y el valor asociado es, a su vez, otra tabla hash en la que cada clave es el nombre de una pestaña modificada siendo el valor asociado la propia pestaña. La ventana principal llama a la **VentanaGuardarSalir** que a la vez que procesa esta tabla hash, construye otra tabla hash equivalente, utilizada para almacenar **JRadioButtons** que permitan al usuario elegir las pestañas modificadas de cualquier estructura que quiere guardar.
  - Una vez que el usuario haya elegido las pestañas que quiere guardar y haya pulsado el botón aceptar, se modifica la tabla hash que contiene las pestañas eliminando de ella las pestañas que no han sido seleccionadas (utilizamos la tabla hash de los **JRadioButtons**). Con esta tabla modificada, la ventana principal recorre las estructuras que contiene, y envía a cada ventana correspondiente la subtabla hash que contiene únicamente las pestañas que tienen que ser guardadas, para guardarlas. Finalmente se cierra la aplicación.
- **Se cierra una ventana**: es un caso particular de cerrar la aplicación, ya que funciona de la misma forma, solo que ahora la ventana que llama a **VentanaGuardarSalir** es la ventana que se cierra. Ahora la tabla hash solo tiene una clave que es el nombre de la estructura de la ventana y cuyo valor asociado es otra tabla hash donde las claves son los nombres de las pestañas

modificadas y cuyos valores asociados son las propias pestañas. El funcionamiento es análogo al de cerrar la aplicación.

6.2.- SUBSISTEMA DE CONTROL DE LA SIMULACIÓN

Para llevar a cabo todo el control sobre las simulaciones, descrito en la **sección 3.4.5**, hemos distinguido los siguientes modos de simulación:

- Ejecución de una o varias operaciones sin animación
- Ejecución de una operación con animación
- Ejecución de varias operaciones con animación

Para saber qué tipo de ejecución se va a realizar, la pestaña que contiene la simulación de la estructura, tiene una serie de métodos y de atributos que explicamos a continuación:

- `int operacionActual`: atributo de tipo entero que indica la posición de la operación actual en el registro de operaciones.
- `int tope`: atributo de tipo entero que indica hasta qué posición, de las operaciones del registro, se va a ejecutar una simulación.
- `boolean pulsadoPlay`: atributo de tipo booleano que indica el estado del botón PLAY/STOP. Si su valor es true, estamos en modo PLAY, y si por el contrario, su valor es false, estamos en modo STOP.
- `public void ejecutarOperaciones()`: método que se encarga de ejecutar todas las operaciones, que hay registradas en el registro de operaciones de la pestaña, desde operacionActual hasta el tope. La ejecución de estas operaciones no tiene animación, es decir, este método se usa cuando utilizamos los controles de simulación INICIO, RETROCEDER, AVANZAR, FIN, o bien cuando se pincha en una de las operaciones del registro.
- `public void leerOperacionSiguiente()`: método que se encarga de ejecutar la siguiente operación a la `operacionActual` de forma que se produzca su animación correspondiente. Este método se usa cuando pulsamos el botón de PLAY. En este caso, vamos a ir ejecutando todas las operaciones, de una en una hasta llegar al final, a no ser que lo paremos antes con el botón STOP. Para

ejecutar una tras otra, cuando finaliza una operación, esta se encarga de llamar, de nuevo, a este método para ejecutar la siguiente.

Tanto `ejecutarOperaciones()` como `leerOperacionSiguiente()`, según la operación que estén procesando, hacen la llamada correspondiente al método que ejecuta esa operación. Para cada operación, se ha creado un método que la ejecuta según el modo de simulación. Para indicar este modo, se le pasan dos parámetros:

- **`boolean`** `animar`: determina si esa operación se tiene que ejecutar con animación o no.
- **`boolean`** `continuarSimulacion`: determina si después de ejecutar esa operación, se van a seguir ejecutando la siguiente.

En función de los valores que tengan estas variables, se puede saber quien ha hecho la llamada. En la ***tabla 7*** se muestra la tabla de verdad de estos parámetros.

animar	continuarSimulacion	Casos
false	false	No se contempla
false	true	La llamada al método la realiza **`ejecutarOperaciones()`**.
true	false	La llamada al método se realiza cuando se pulsa sobre una operación de la estructura.
true	true	La llamada al método la realiza **`leerOperacionSiguiente()`**.

Tabla 7 – Tabla de verdad de animar y continuarSimulacion

El esquema general de los métodos que ejecutan las operaciones de las estructuras de datos es:

```
public void operacion(boolean animar, boolean continuarSimulacion, String dato) {
   try {

      // Se realiza la operación en la parte de las implementaciones.
      ED.operacion(dato);

      // Se añade un botón con la operación en el registro de operaciones
      // si se ha pulsado el botón de ejecutar la operación.
      if (!continuarSimulacion)
```

```
            añadirOperacion("Operacion " + dato);

        …
        // se activan o desactivan los botones de tal forma que no
        // se permita que se ejecuten operaciones que den errores.
        …

        // Indicamos en el cuadro de diálogo la operación que se ha ejecutado.
        setTextoCuadroDialogo("Operacion " + dato, false);

        // Actualizamos la posición de la flecha que apunta a la operación actual.
        avanzarFlecha(animar);

        if (animar){
            // Enviamos el vector de acciones, construido en la implementación, a la
            // estructura de datos gráfica. También mandamos continuarSimulacion.
            EDGrafica.setValoresOperacion(ED.getVectorAcciones(), continuarSimulacion);

            // Mandamos a la EDGrafica que procese cada una de las acciones del
            // vector, realizando sus respectivas animaciones.
            EDGrafica.ejecutarAcciones();
        }
        else
            // Si animar es false EDGrafica no dibuja nada
            setTextoCuadroDialogo("Se ha realizado la Operación", false);
    }
    catch (Exception e) {
        e = new Exception("Error en Operacion");
    }
}
```

Como podemos comprobar, animar y continuar simulación, rigen la forma de ejecutar las operaciones.

Las operaciones se realizan con o sin animación gracias a que la parte gráfica de las estructuras de datos pueden dibujar de 2 formas:

- *A la* ***EDGrafica*** *se le pasa el vector de acciones de la operación y se manda ejecutar dicho vector a partir del último estado de la estructura de datos. Así conseguimos la animación de la operación.*
- *Si queremos dibujar una estructura de datos sin animar ninguna operación, la* ***EDGrafica*** *nos permite dibujar dicha estructura a partir de un iterador que contiene, en el orden adecuado, los elementos. (Es para tener este iterador por lo que todas las implementaciones de las estructuras heredan de* **AbstractList**, *e implementan los métodos* `get()` *y* `size()`)

  Es el método `ejecutarOperaciones()` *el encargado de, después de ejecutar todas las operaciones desde operacionActual hasta tope sin animación, llamar a EDGrafica para que dibuje la estructura de datos a partir de su iterador.*

En la **tabla 8** podemos ver el funcionamiento interno de los distintos controles de simulación.

Clic sobre operación de la estructura	`pulsadoPlay = false` `operación(true, false)`

 **Inicio**	`if (operacionActual != 0){`
	`  tope = 0`
	`  operacionActual = -1`
	`  ejecutarOperaciones()`
	`}`

 **Retroceder**	`if (operacionActual != 0){`
	`  tope = operacionActual - 1`
	`  operacionActual = 0`
	`  ejecutarOperaciones()`
	`}`

 **Play**	`if (operacionActual != posición última operación del registro){`
	`  pulsadoPlay = true`
	`  leerSiguienteOperacion()`
	`}`

 **Stop**	`pulsadoPlay = false`

 **Avanzar**	`if (operacionActual ¡= posición última operación del registro){`
	`  tope = operacionAcutal +1`
	`  ejecutaOperaciones()`
	`}`

 **Fin**	`if (operacionActual != posición última operación del registro){`
	`  tope = posición última operación del registro`
	`  ejecutaOperaciones()`
	`}`

Clic sobre operación del registro	`if (posición operación del registro elegida < operacionActual)`
	`  operacionActual = -1`
	`tope = posición operación del registro elegida`
	`ejecutaOperaciones()`

Tabla 8 – Detalles de implementación de los controles de simulación

Para entender el funcionamiento interno de los controles de simulación, a continuación se muestran los diagramas de secuencia para los modos de simulación posibles.

6.2.1.- Diagramas de Secuencia

Como hemos comentado anteriormente, hay tres tipos distintos de simulación, para cada uno de los cuales, se ha realizado su diagrama de secuencia correspondiente.

- **Ejecución de una o varias operaciones sin animación**: este tipo de simulación, que corresponde a la pulsación de los botones INICIO, RETROCEDER, AVANZAR, FIN o cualquiera de los botones del registro de operaciones, sigue una secuencia como la del diagrama de la **figura 146**.
- **Ejecución de una operación con animación**: este tipo de simulación, que corresponde a la pulsación de un botón de una operación, de la estructura de datos que se está simulando. se explica mediante el diagrama de la **figura 147**.
- **Ejecución de varias operaciones con animación**: este tipo de simulación, que corresponde a la pulsación del botón PLAY. se explica mediante el diagrama de la **figura 148**.

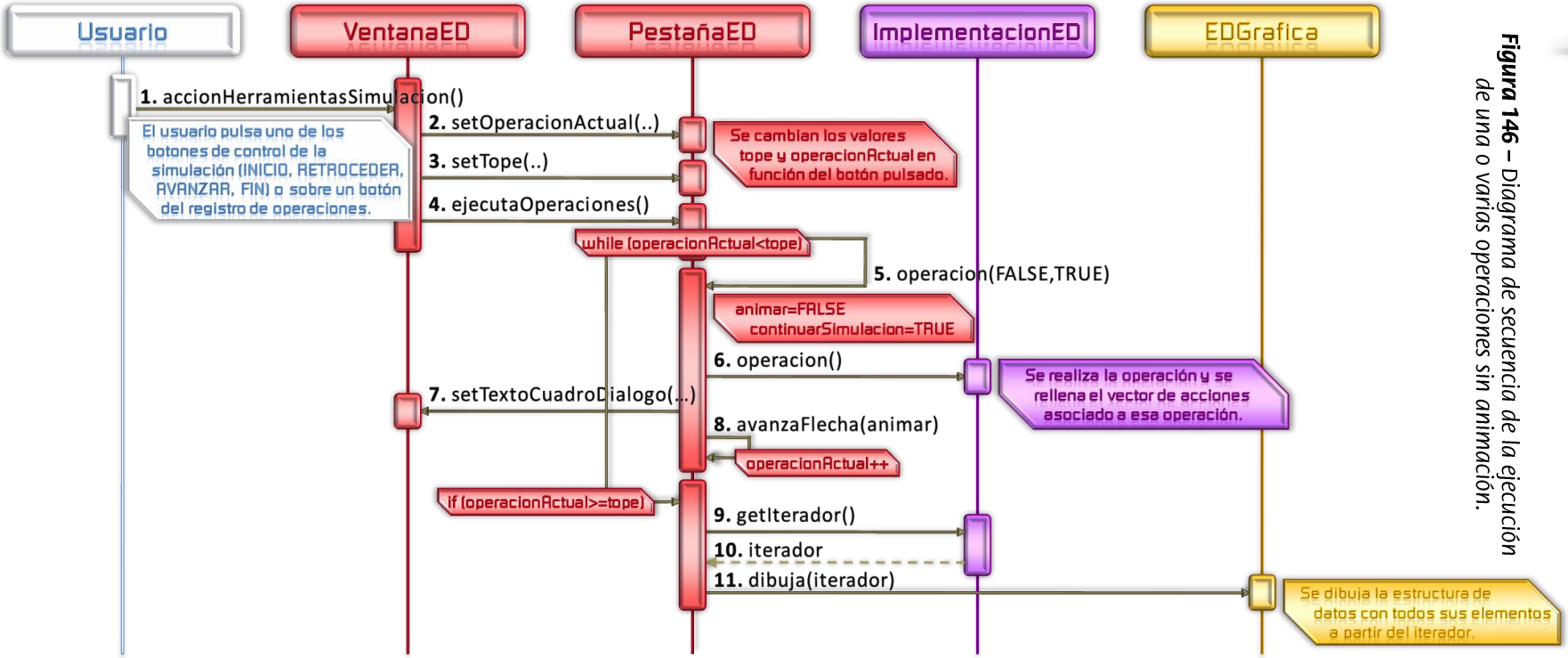


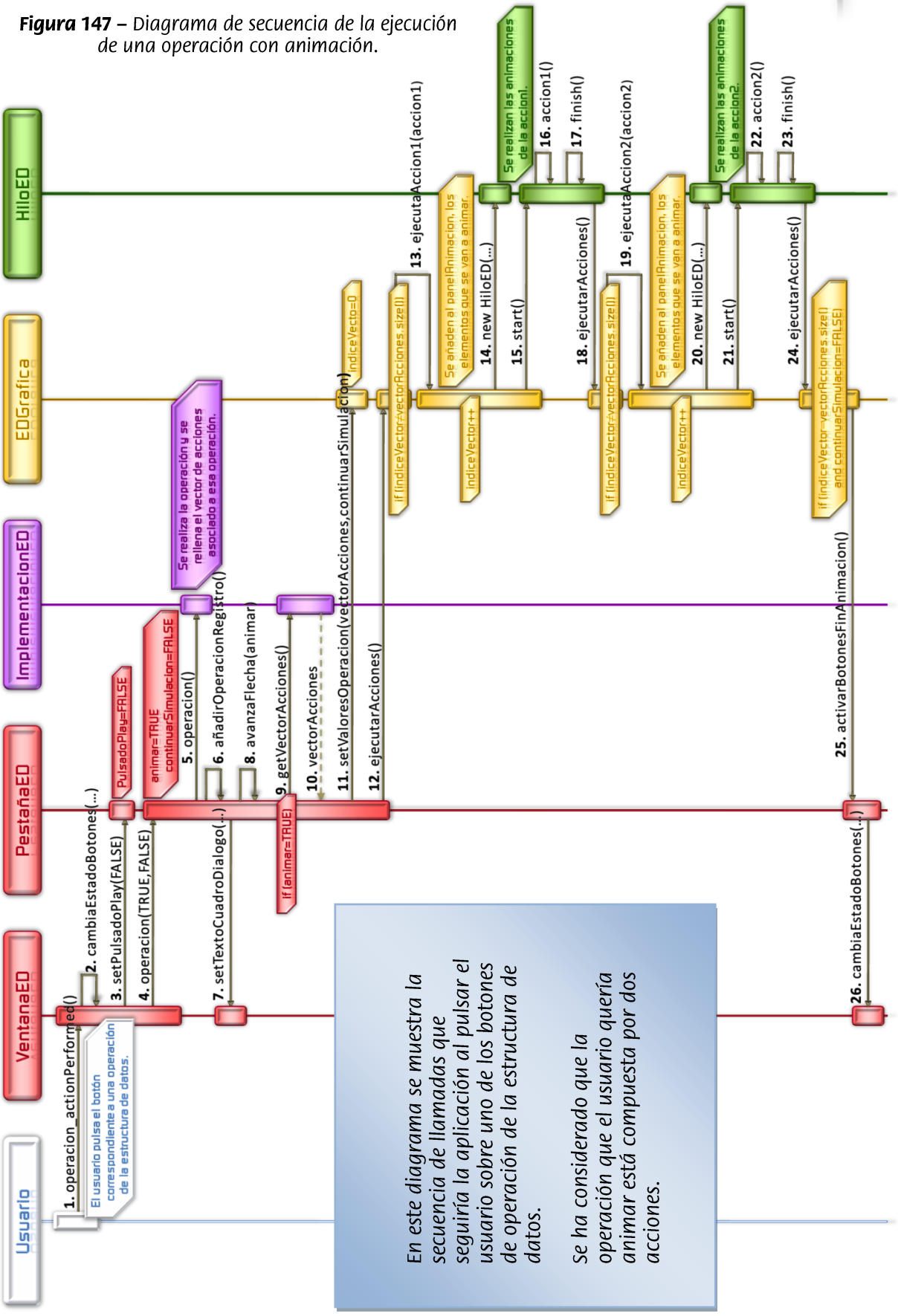
Figura 146 – *Diagrama de secuencia de la ejecución de una o varias operaciones sin animación.*

En este diagrama se muestra la secuencia de llamadas que seguiría la aplicación al pulsar el usuario alguno de los botones INICIO, RETROCEDER, AVANZAR, FIN *o cualquiera de los botones del registro de operaciones.*

La diferencia entre unos y otros estriba en el valor que se le asigna a operacionActual *y a* tope.

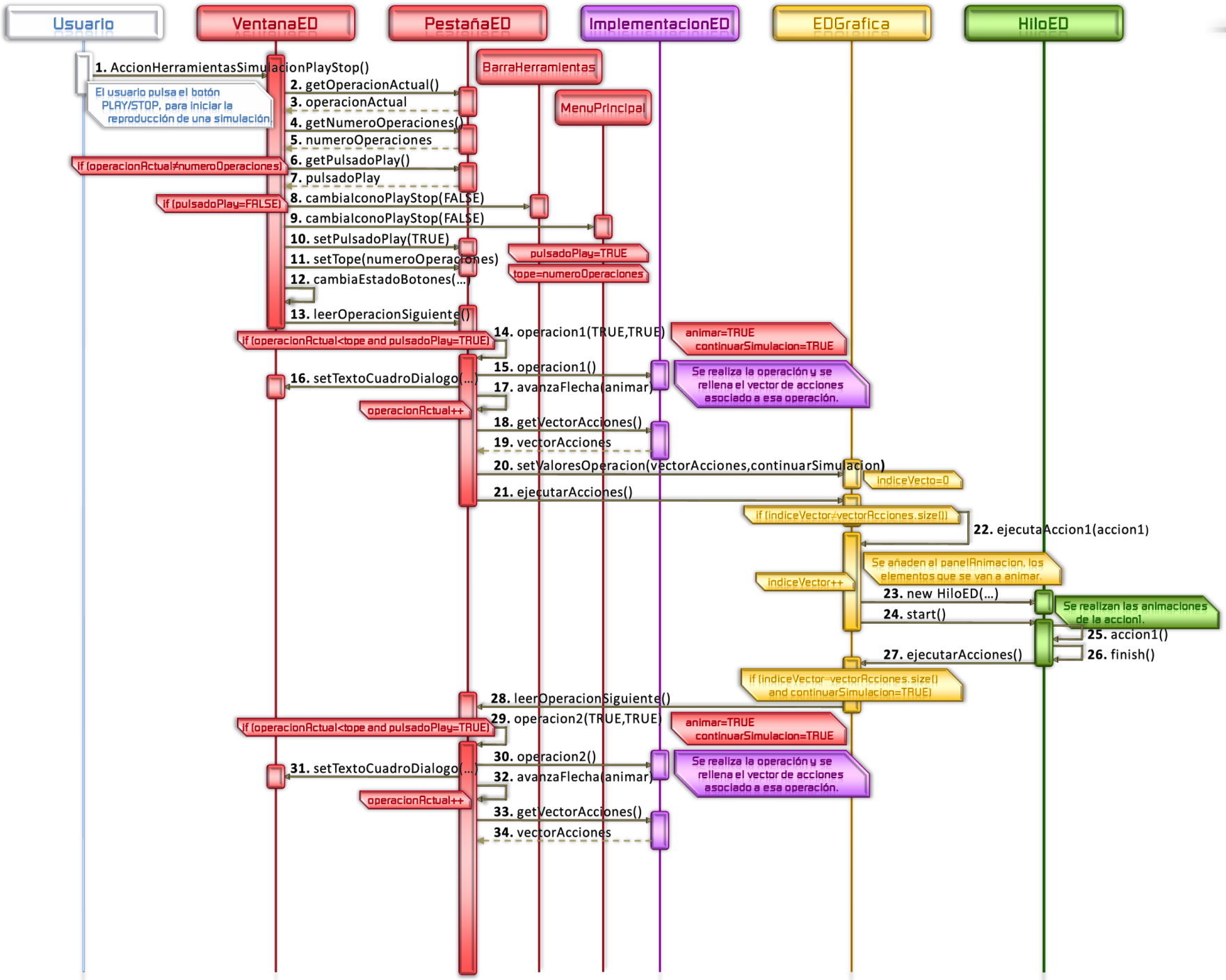
Hacemos notar que el funcionamiento de estos botones, se basa en la idea de obtener y dibujar los elementos de una estructura de datos en un momento concreto de la simulación.

Figura 147 – *Diagrama de secuencia de la ejecución de una operación con animación.*



En este diagrama se muestra la secuencia de llamadas que seguiría la aplicación al pulsar el usuario sobre uno de los botones de operación de la estructura de datos.

Se ha considerado que la operación que el usuario quería animar está compuesta por dos acciones.


Usuario
VentanaED
PestañaED
ImplementacionED
EDGrafica
HiloED
BarraHerramientas
MenuPrincipal
1. AccionHerramientasSimulacionPlayStop()
El usuario pulsa el botón PLAY/STOP, para iniciar la reproducción de una simulación
2. getOperacionActual()
3. operacionActual
4. getNumeroOperaciones()
5. numeroOperaciones
6. getPulsadoPlay()
7. pulsadoPlay
if (operacionActual≠numeroOperaciones)
if (pulsadoPlay=FALSE)
8. cambiaIconoPlayStop(FALSE)
9. cambiaIconoPlayStop(FALSE)
10. setPulsadoPlay(TRUE)
pulsadoPlay=TRUE
11. setTope(numeroOperaciones)
tope=numeroOperaciones
12. cambiaEstadoBotones(...)
13. leerOperacionSiguiente()
if (operacionActual<tope and pulsadoPlay=TRUE)
14. operacion1(TRUE,TRUE)
animar=TRUE
continuarSimulacion=TRUE
15. operacion1()
Se realiza la operación y se rellena el vector de acciones asociado a esa operación.
16. setTextoCuadroDialogo(...)
17. avanzaFlecha(animar)
operacionActual++
18. getVectorAcciones()
19. vectorAcciones
20. setValoresOperacion(vectorAcciones,continuarSimulacion)
indiceVecto=0
21. ejecutarAcciones()
if (indiceVector≠vectorAcciones.size())
22. ejecutaAccion1(accion1)
Se añaden al panelAnimacion, los elementos que se van a animar.
indiceVector++
23. new HiloED(...)
24. start()
Se realizan las animaciones de la accion1.
25. accion1()
26. finish()
27. ejecutarAcciones()
if (indiceVector=vectorAcciones.size() and continuarSimulacion=TRUE)
28. leerOperacionSiguiente()
if (operacionActual<tope and pulsadoPlay=TRUE)
29. operacion2(TRUE,TRUE)
animar=TRUE
continuarSimulacion=TRUE
30. operacion2()
Se realiza la operación y se rellena el vector de acciones asociado a esa operación.
31. setTextoCuadroDialogo(...)
32. avanzaFlecha(animar)
operacionActual++
33. getVectorAcciones()
34. vectorAcciones

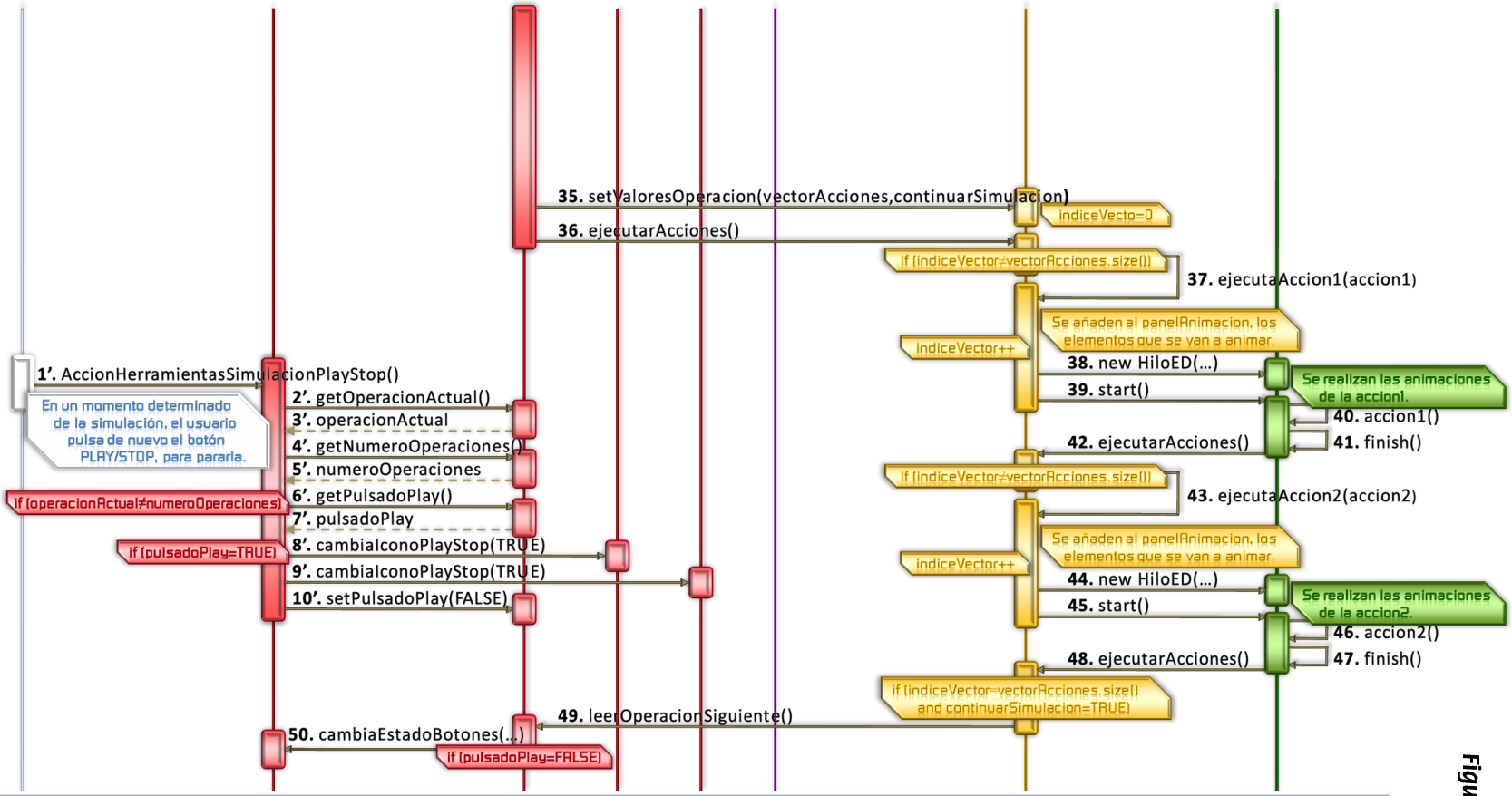


Figura 148 – *Diagrama de secuencia de la ejecución de varias operaciones con animación.*

En este diagrama se muestra la secuencia de llamadas que seguiría la aplicación al pulsar el usuario sobre el botón PLAY y más tarde sobre el botón STOP.

El efecto del botón PLAY es animar todas las operaciones desde la operación actual, hasta la última operación, a no ser que el usuario pare la simulación pulsando el botón STOP.

En el diagrama se muestra como la primera operación está compuesta por una única acción mientras que la segunda operación se desglosa en dos acciones.

Mientras se está ejecutando la segunda operación, se puede observar que aparece una nueva secuencia que está originada por la pulsación del botón STOP por parte del usuario. Este es el motivo por el cual no se continúa con la animación de más operaciones en la primera secuencia.

Hemos hecho estos diagramas de manera general, es decir para cualquier estructura y simulando cualquier operación. Para entender los diagramas que se muestran a continuación, hay que tener en cuenta que:

- Con "operacion" nos referimos a cada una de las operaciones concretas de cada estructura de datos. Así,
  - Si estamos simulando una pila, tendríamos que operación puede ser Crear, Apilar, Desapilar, Cima o EsVacia?.
  - Si estamos simulando una cola, tendríamos que operación puede ser Crear, PedirVez, Avanzar, Primero o EsVacia?.
  - Si estamos simulando un árbol binario de búsqueda, tendríamos que operación puede ser Crear, Plantar, Insertar, Eliminar, Esta?, Minimo?, Maximo?, Raiz?, HijoIzquierdo?, HijoDerecho?, Altura, EsVacio?, Preorden, Inorden o Postorden.

  Para no complicar los diagramas, se han considerado operaciones que no necesitan datos de entrada. Si la operación requiriera de algún dato habría que tratarlo de la forma adecuada.
- Con "accion" nos referimos a cada una de las acciones en las que se desglosa una operación. Cada operación de las estructuras de datos está compuesta por una o más acciones (ver **sección 6.4**)
- Con "VentanaED" nos referimos a cada una de las ventanas de estructuras de datos, que pueden ser: VentanaPila, VentanaCola o VentanaArbolBinarioBusqueda.
- Con "PestañaED" nos referimos a cada una de las pestañas que están contenidas en las ventanas de estructuras de datos. Estas pueden ser: PestañaPila, PestañaCola o PestañaArbolBinarioBusqueda.
- Con "ImplementacionED" nos referimos a cada una de las implementaciones de las estructuras de datos. Estas pueden ser: ImplemetacionPila, ImplementacionCola o ImplementacionArbolBinarioBusqueda.
- Con "EDGrafica" nos referimos a la parte gráfica de la aplicación, es decir, la que implementa y dibuja los elementos de las estructuras de datos. "EDGrafica" depende de la estructura de datos y la visualización que se está simulando. Puede ser:
  - PilaGraficaUsuario, PilaGraficaEstatica o PilaGraficaDinamica

- ColaGraficaUsuario, ColaGraficaEstatica o ColaGraficaDinamica
- ArbolBinarioBusquedaGraficoUsuario.

- Con "HiloED" nos referimos a las hebras que realizan las simulaciones de las acciones. "HiloED" depende de la estructura de datos y la visualización que se está simulando. Puede ser:
  - HiloPilaUsuario , HIloPilaEstatica o HiloPilaDinamica
  - HiloColaUsuario, HiloColaEstatica o HiloColaDinamica
  - HiloArbolBinarioBusquedaUsuario
- "AccionHerramientaSimulacion()" puede ser:
  - AccionHerramientasSimulacionInicio() si se ha pulsado INICIO
  - AccionHerramientasSimulacionRetroceder() si se ha pulsado RETROCEDER
  - AccionHerramientasSimulacionAvanzar() si se ha pulsado AVANZAR
  - AccionHerramientasSimulacionFin() si se ha pulsado FIN
  - textOperacion_actionPerformed() si se ha pulsado sobre una operación del registro

6.3.- Posición y Tamaño de las Estructuras de Datos

La representación gráfica de las estructuras de datos, que se visualiza en el **panelAnimacion** de las pestañas, se construye a partir de un punto de referencia. Esta posición de referencia dependerá del tamaño del **panelAnimacion** y de la forma que va a tener la representación de la estructura de datos.

La acción de mostrar u ocultar alguna de las componentes de la interfaz grafica (opciones disponibles en el menú Ver), provoca que se redimensione el **panelAnimacion**. Se ha implementado un mecanismo por el cual, cuando sucede esto, se recalcula la posición de referencia de la estructura de datos, haciendo que la estructura se coloque acorde con el nuevo tamaño del panel.

Uno de los mayores problemas al representar una estructura de datos es la limitación de espacio fijado por el tamaño de las pantallas.

En las versiones anteriores de **VEDYA** se había intentado solucionar esta limitación, añadiendo al panel, donde se representan las estructuras, un **JScrollBar**.

Pero esta solución no dio un resultado satisfactorio, ya que el comportamiento de la visualización gráfica de la estructura de datos, cuando se sobrepasaba el espacio físico de la pantalla destinado a este fin, era, en ocasiones, bastante extraño. Por ejemplo veamos cómo se comportaban las colas en la visualización de usuario.

En la **figura 149** se puede ver que se ha creado una cola y se han insertado los elementos 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 y 11. Hasta aquí no hay ningún problema y la estructura se visualiza correctamente.

En la **figura 150** mostramos como quedaría la cola si le añadiésemos el elemento 12. Se puede observar que nos hemos movido con la barra de desplazamiento a la izquierda y no vemos el último elemento añadido, además no se muestra el primer elemento, ni el surtidor.

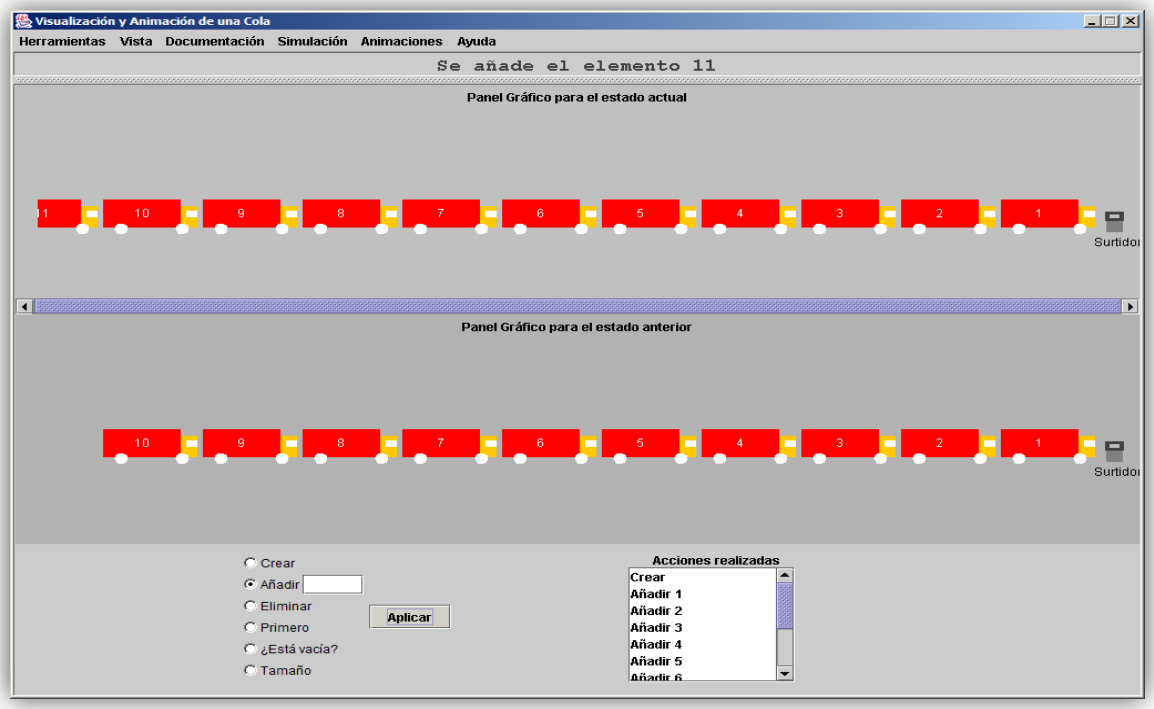


Figura 149 – Cola en vista de usuario con 11 elementos (Versión anterior)

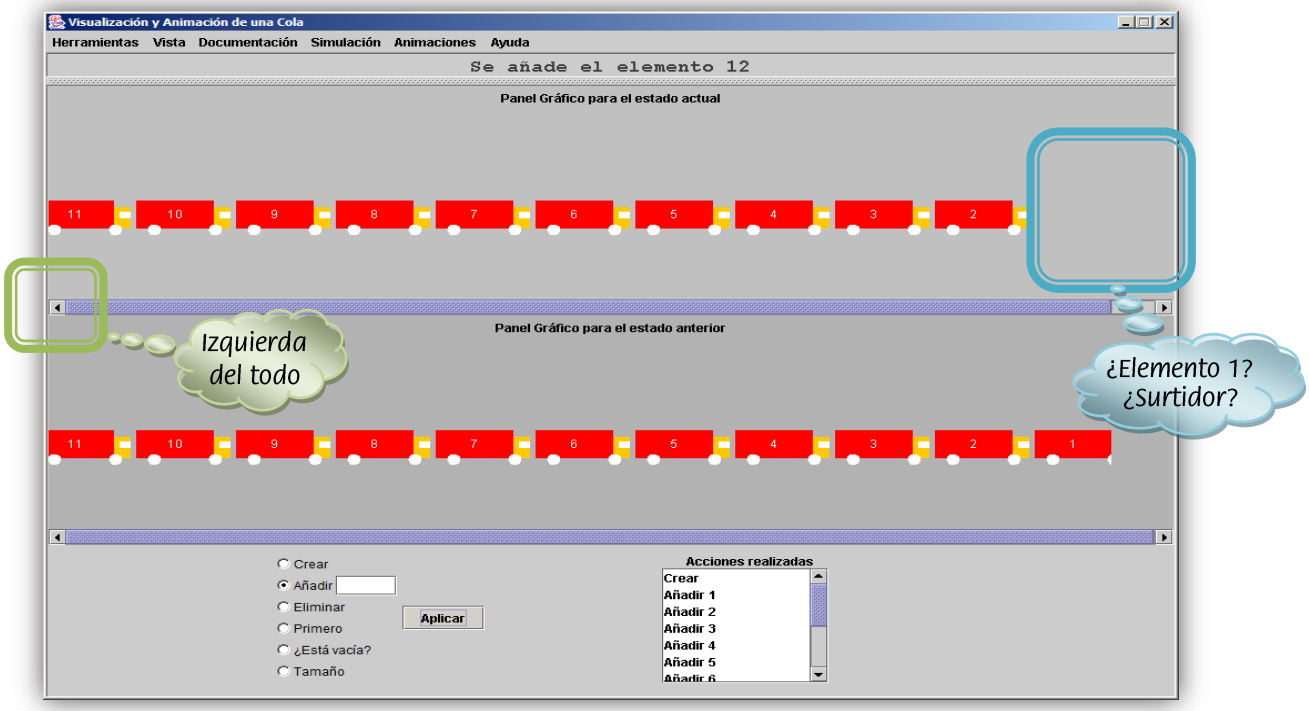


Figura 150 – *Cola en vista de usuario con 12 elementos (Versión anterior)*

En la versión de **VEDYA** *que hemos desarrollado, como se ha comentado con anterioridad, para las estructuras sencillas como las pilas y las colas, se ha solucionado el problema del espacio, limitando el número de elementos de las estructuras de datos (este número se ha elegido lo suficientemente grande como para que no dé problemas en el aspecto didáctico de la herramienta) y distribuyéndolos por toda la zona de la pantalla destinada a la representación gráfica de la estructura.*

Esta solución no nos parecía suficiente para estructuras más complejas, como los árboles binarios de búsqueda. Por ejemplo, un árbol con pocos elementos puede ocupar mucho espacio si la profundidad es suficientemente grande. Este motivo ha sido la razón por la cual hemos considerado que el espacio disponible no podía poner límites a las estructuras, y se han implementado los siguientes controles:

- ***Controles de posición:*** *en cada uno de los puntos cardinales del* ***panelAnimacion*** *de cada pestaña, se colocan botones que controlan la posición de la estructura. Estos botones hacen variar la posición de referencia, que hemos comentado anteriormente, haciendo que la estructura se mueva por la pantalla. En la* ***figura 151****, se muestra el esquema de funcionamiento de estos*

controles, en el que suponemos que v es la variación en pixels que se produce al pulsar cada botón.

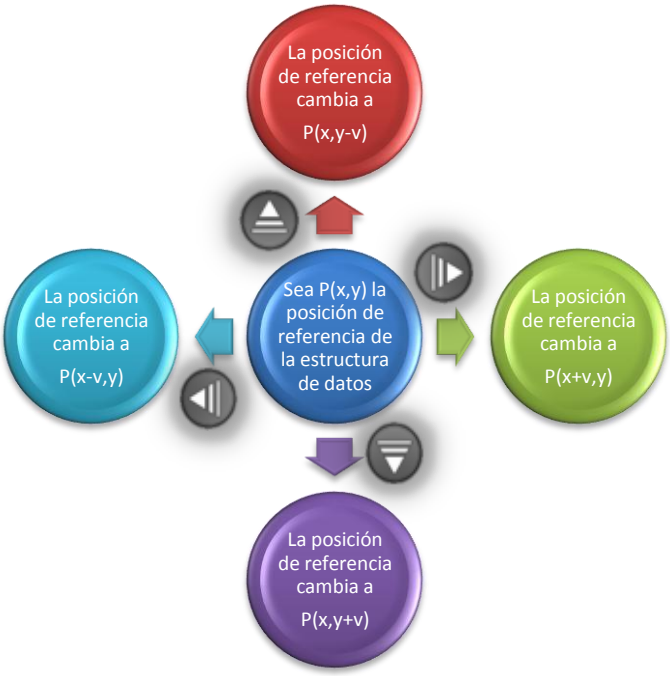


Figura 151 – Diagrama de funcionamiento de los controles de posición

- **Controles de tamaño:** en la esquina superior derecha del **panelAnimacion** de cada pestaña hay colocados 4 botones que controlan el tamaño de las estructuras arbóreas. En las estructuras de este tipo se han definido los siguientes valores (ver **figura 152**):
  - **Separación vertical**: es la separación que hay entre un nivel y el nivel inmediatamente inferior.
  - **Separación horizontal**: es la separación que hay entre los elementos del último nivel. Es decir la separación entre los elementos hoja.

  Mediante los controles de tamaño, podemos aumentar o disminuir las separaciones vertical y horizontal.

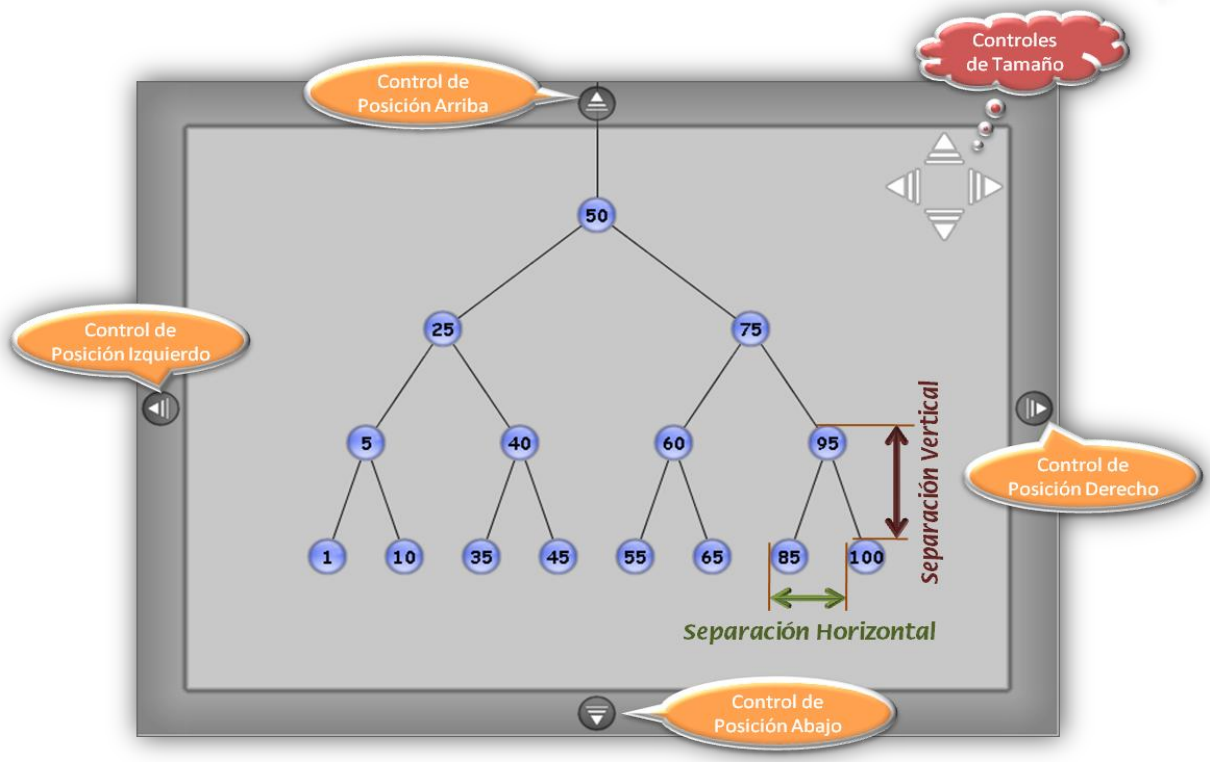


Figura 152 – Definición de Separación horizontal y separación vertical

6.4.- Vector Acciones

El vector de acciones se rellena con las unidades mínimas en las que se puede desglosar una operación de la estructura de datos. A estas unidades mínimas las hemos llamado acciones, y deben contener toda la información necesaria para que la parte grafica de la herramienta sepa como dibujarlas y que animaciones realizar.

En las versiones anteriores de **VEDYA**, el vector de acciones seguía las siguientes consideraciones:

- Dependiendo de la estructura de datos, las acciones podían ser simples o compuestas.
  - Acción simple era aquella acción que no requería una composición de animaciones. Por ejemplo, una acción simple sería la inserción de un elemento en la cola.
  - Acción compuesta era aquella acción que constará de varias animaciones independientes. Por ejemplo, la inserción en un árbol AVL requeriría de

una acción compuesta ya que además de insertar hay que equilibrar. Por tanto, además de pasarle las acciones necesarias para la inserción, habría que pasarle las acciones necesarias para el equilibrio, que a su vez son acciones compuestas.

- Al panel de dibujo se le pasaba el vector de acciones. Este panel era el intermediario en la comunicación entre la interfaz y la parte gráfica, únicamente cuando era necesario; por ejemplo, cuando se efectuaba una operación consultora como podía ser preguntar si la pila es vacía, no era necesario.
- La información de los vectores de acciones, en el caso de las estructuras de datos, se rellenaba en la interfaz.

En la nueva versión de **VEDYA** nos propusimos realizar animaciones más elaboradas, por lo que necesitábamos que la implementación de las estructuras nos proporcionara más información. Este ha sido el motivo por el cual hemos cambiado algunas de las consideraciones a tener en cuenta por el vector de acciones.

En la actualidad, el vector de acciones se rellena en la implementación de las estructuras de datos con las acciones en las que se desglosan las operaciones.

Hay operaciones muy sencillas, como por ejemplo las de las pilas y las colas, en las que una operación solo afecta a un elemento y siempre se realiza del mismo modo, siendo indistinto el estado en el que se encuentre la estructura de datos. Sin embargo, hay otras más complicadas donde la operación afecta a más de un elemento de la estructura y su ejecución no siempre se realiza del mismo modo ya que depende del estado de la estructura. Esto ocurre, por ejemplo, al eliminar un elemento del árbol binario de búsqueda. La ejecución de la operación será distinta si el elemento existe o no, si es una hoja o si tiene un solo hijo o si tiene dos hijos. Las operaciones sencillas se pueden describir mediante una única acción, mientras que las operaciones más complejas están compuestas por varias acciones. En el primer caso el vector contendría una sola acción y en el segundo caso el vector de acciones contendría todas las acciones necesarias para poder representar la animación. Es decir, todas las operaciones se dividen en acciones atómicas.

Hemos querido separar por completo lo que son las operaciones, de lo que son las acciones. Las acciones no tienen el mismo nombre que las operaciones, aunque éstas estén formadas por una única acción. La operación apilar el 7 no tiene toda la información que tiene la acción añadir el 7, ya que la acción contiene atributos auxiliares que indican cómo y dónde se tiene que realizar la acción.

Para que la información que se ha guardado en las acciones sea útil para las animaciones, tiene que haber una correspondencia entre los elementos de la parte de la implementación y los de la parte grafica. Es decir, los elementos que intervienen en la parte de la implementación de las estructuras de datos, no son los mismos objetos que usamos para dibujarlos, por lo que hemos de tener algún mecanismo para identificar unos con otros. Para ello, en la implementación de las estructuras, sea cual sea la implementación utilizada, cada elemento se numera, bien sea en un array o en una estructura con punteros. Haciendo una pequeña traducción de la numeración del elemento, podemos conseguir identificar su correspondiente elemento gráfico en la parte gráfica de la aplicación. Esta traducción, para cada una de las estructuras, se explicará en la **sección 6.6**.

Ya que cada estructura está implementada de distinta forma y se necesita distinta información para representarla, se han creado acciones personalizadas para cada una de las estructuras de datos. Se ha realizado el esfuerzo de unificar todas las acciones de una misma estructura en una única clase, de manera que todas las acciones atómicas de una misma estructura poseerán los mismos atributos (aunque algunas acciones no los utilicen. Para ello se han implementado las clases del paquete utilidades que se detallan a continuación.

6.4.1.- Acción Pila

Las acciones de las pilas contienen la siguiente información:

- **String acción**: identifica la acción atómica a realizar. Los posibles valores de este String son:
  - Crea: acción implicada en la operación Crear.
  - Añade: acción implicada en la operación Apilar.
  - Elimina: acción implicada en la operación Desapilar.

- Señala: acción implicada en la operación Cima.
- EsVacia?: acción implicada en la operación EsVacia?.

Como hemos comentado anteriormente, los nombres de las acciones generalmente no coinciden con los nombres de las operaciones ya que queremos que se distinga lo que es una acción de lo que es una operación.

- **Object dato**: identifica el dato sobre el que se realizará la acción.
- **int posicionDato**: indica la posición, en la estructura de PilaGrafica, donde se encuentra el elemento gráfico correspondiente al dato. Para cada una de las acciones el valor de posicionDato es:
  - Crea: posicionDato = -1. No importa el valor ya que no se realiza acción sobre ningún elemento, sino sobre la estructura.
  - Añade: posicionDato = posición de la estructura de PilaGrafica donde se va a añadir el elemento.
  - Elimina: posicionDato = posición de la estructura de PilaGrafica donde está el elemento que se va a eliminar.
  - Señala: posicionDato = posición de la estructura de PilaGrafica donde está el elemento que se va a señalar.
  - EsVacia: posicionDato = posición de la estructura de PilaGrafica donde se encuentra el último elemento de la pila.
- **int posicionDatoAnterior**: indica la posición anterior en la estructura de PilaGrafica, del elemento sobre el que vamos a realizar al acción.

6.4.2.- Acción Cola

Las acciones de las colas contienen la siguiente información:

- **String acción**: Identifica la acción atómica a realizar. Los posibles valores de este String son:
  - Crea: acción implicada en la operación Crear.
  - Añade: acción implicada en la operación PedirVez.
  - Elimina: acción implicada en la operación Avanzar.
  - Señala: acción implicada en la operación Primero.
  - EsVacia?: acción implicada en la operación EsVacia?.

Los nombres de las acciones generalmente no coinciden con los nombres de las operaciones ya que queremos que tengan nombres genéricos.

- **Object dato**: Identifica el dato sobre el que se realizara la acción.
- **int posicionDato**: Indica en qué posición de la estructura de la ColaGrafica se realizará la acción. Para cada una de las acciones el valor de posicionDato es:
  - Crea: posicionDato = -1. No importa el valor ya que no se realiza acción sobre ningún elemento, sino sobre la estructura.
  - Añade: posicionDato = posición, en la estructura de ColaGrafica, donde se va a añadir el elemento.
  - Elimina: posicionDato = posición, en la estructura de ColaGrafica, donde está el elemento que se va a eliminar.
  - Señala: posicionDato = posición, en la estructura de ColaGrafica, donde está el elemento que se va a señalar.
  - EsVacia: posicionDato = posición, en la estructura de ColaGrafica, donde se encuentra el primer elemento de la cola.
- **int posicionDatoAnterior**: Indica la posición anterior en la estructura de la ColaGrafica del elemento sobre el que vamos a realizar al acción. Para cada una de las acciones el valor de posicionDatoAnterior es:
  - Crea: posicionDatoAnterior = -1. No importa el valor ya que no se realiza acción sobre ningún elemento, sino sobre la estructura.
  - Añade: posicionDatoAnterior = posición, en la estructura de ColaGrafica, del elemento anterior donde se va a añadir el nuevo elemento.
  - Elimina: posicionDatoAnterior = posición, en la estructura de ColaGrafica, del elemento siguiente al elemento que se va a eliminar.
  - Señala: posicionDatoAnterior = -1. No importa el valor ya que no es necesaria la posición del elemento anterior para realizar esta acción.
  - EsVacia: posicionDatoAnterior = posición, en la estructura de ColaGrafica, donde se encuentra el último elemento de la cola.
- **int tamañoCola**: Indica el tamaño de la cola después de realizarse la acción.

6.4.3.- Acción Árbol Binario de Búsqueda

Las acciones de los árboles binarios de búsqueda contienen la siguiente información:

- **String acción**: Identifica la acción atómica a realizar. Los posibles valores de este String son:
  - Crea: única acción atómica implicada en la operación Crear.
  - Planta: única acción atómica implicada en la operación Plantar.
  - Añade: única acción atómica implicada en la operación Insertar.
  - BuscaElimina, Elimina, SubeNivel, BuscaMinimoHijoDerecho e Intercambia: son las acciones implicadas en la operación Eliminar.
    La operación Eliminar siempre va a empezar con la acción BuscaElimina (Busca el elemento que queremos eliminar):
    - Si no se encuentra el elemento no se hace nada más (Dato = null)
    - Si el elemento que se va a eliminar es una hoja, se manda además la acción Elimina.
    - Si el elemento que se va a eliminar no tiene hijo izquierdo o no tiene hijo derecho (o exclusiva), se manda además la acción SubeNivel.
    - Si el elemento que se va a eliminar tiene hijo izquierdo e hijo derecho, se mandan además la acciones BuscaMinimoElimina, Intercambia y SubeNivel.
  - Preorden: única acción atómica implicada en la operación Preorden.
  - Inorden: única acción atómica implicada en la operación Inorden.
  - Postorden: única acción atómica implicada en la operación Postorden.
  - Raiz?: única acción atómica implicada en la operación Raiz.
  - HijoIzquierdo?: única acción atómica implicada en la operación HijoIzquierdo.
  - HijoDerecho?: única acción atómica implicada en la operación HijoDerecho.
  - EsVacio?: única acción atómica implicada en la operación EsVacio.
  - Esta?: única acción atómica implicada en la operación Esta.
  - Maximo?: única acción atómica implicada en la operación Maximo.
  - Minimo?: única acción atómica implicada en la operación Minimo.

  Los nombres de las acciones no coinciden con los nombres de las operaciones ya que queremos distinguir lo que son operaciones de lo que son acciones.
- **Object dato**: Identifica el dato sobre el que se realizará la acción.

- **Int[] caminoAccion**: contiene la secuencia de posiciones de nodos que van a intervenir en la operacion. Para cada una de las acciones, este array va a guardar lo siguiente:
  - Crea: Array vacío (no lo usamos).
  - Planta: Array con la raíz del hijo izquierdo y la raíz del hijo derecho.
  - Añade: Array con los nodos del árbol que se tienen que ir recorriendo hasta llegar a la posición donde se añade el elemento.
  - BuscaElimina: Array con los nodos del árbol que se tienen que ir recorriendo hasta llegar a la posición del elemento que se quiere eliminar.
  - Elimina: Array vacío (no lo usamos).
  - SubeNivel: Array con el subarbol que vamos a subir un nivel.
  - BuscaMinimoElimina: Array con los nodos del arbol (no tienen por qué empezar en la raíz del árbol principal) que se tienen que recorrer hasta llegar al mínimo del hijo derecho.
  - Intercambia: Array con los dos elementos que se van a intercambiar (el primero es el elemento que se va a eliminar y el segundo es mínimo del hijo derecho).
  - Preorden: Array con la secuencia de nodos que se tienen que ir recorriendo para realizar un recorrido en Preorden.
  - Inorden: Array con la secuencia de nodos que se tienen que ir recorriendo para realizar un recorrido en Inorden.
  - Postorden: Array con la secuencia de nodos que se tienen que ir recorriendo para realizar un recorrido en Postorden.
  - Raiz?: Array vacío (no lo usamos).
  - HijoIzquierdo?: Array con el subárbol (hijo izquierdo) que queremos señalar.
  - HijoDerecho?: Array con el subárbol (hijo derecho) que queremos señalar.
  - EsVacio?: Array vacío (no lo usamos).
  - Esta?: Array con los nodos del árbol (empieza siempre en la raíz del árbol principal) que se tienen que recorrer hasta llegar al elemento buscado.
  - Maximo?: Array con los nodos del árbol (empieza siempre en la raíz del árbol principal) que se tienen que recorrer hasta llegar al elemento Máximo del árbol.

- Minimo?: Array con los nodos del árbol (empieza siempre en la raíz del árbol principal) que se tienen que recorrer hasta llegar al elemento Mínimo del árbol.

- ***int altura***: Altura del árbol después de realizar la acción.
- ***boolean éxito***: Indica si la acción se ha realizado con éxito o no, es decir, en las acciones de búsqueda devuelve true si se ha encontrado el elemento que buscábamos y false si no lo hemos encontrado.

6.5.- Cómo se dibuja

En esta nueva versión de **VEDYA** se ha cambiado la filosofía de cómo dibujar las estructuras de datos.

En versiones anteriores, se representaban las estructuras de datos dibujando sus elementos a partir de figuras geométricas (rectas, cuadrados, círculos, elipses, ...) teniendo que implementar el método paint() de los paneles de dibujo.

Durante el desarrollo de este proyecto, hemos logrado una mayor vistosidad de la aplicación gracias a la utilización de imágenes externas. Nos hemos servido de las propiedades de JComponent (en concreto JLabel) para representar tanto los contenedores de las estructuras como sus elementos:

- En las visualizaciones de las estructuras, que para ser representadas requieren el dibujo de un contenedor, se ha utilizado un JLabel que contiene la imagen del contenedor de la estructura de datos. Este JLabel se posiciona centrado en el panel de animación.
- Para representar los elementos de las estructuras de datos se han utilizado objetos de la clase ElementoGrafico. Para dibujar estos elementos se hace una composición de un texto (JLabel, que contiene el valor del elemento) y una imagen (JLabelElemento, que contiene la imagen gráfica que representa el elemento).

  Para cada una de las visualizaciones de cada una de las estructuras de datos la clase ElementoGrafico se ha especializado ya que cada visualización requiere de elementos y acciones diferenciados. Resultando:

  - ElementoGraficoPilaUsuario

- ElementoGraficoColaUsuario
- ElementoGraficoABBUsuario
- ElementoGraficoPilaEstatica
- ElementoGraficoColaEstatica
- ElementoGraficoPilaDinamica
- ElementoGraficoColaDinamica

Hemos utilizado JComponent's por lo siguiente.

- El panelAnimacion es un **JPanel** que a su vez es un Container que almacena **JComponent's**. El método repaint() de un **JPanel** llama automáticamente a los métodos repaint() de cada uno de los componentes que contiene. Conocido este funcionamiento, nos ha resultado mucho más sencillo realizar las animaciones sobre las estructuras de datos, ya que tratamos los elementos gráficos de las estructuras de forma totalmente independiente unos de otros.
- Al panel se le ha aplicado el **XYLayout** (propio de las librerías de JBuilder) que nos da la posibilidad de posicionar cada uno de los elementos por coordenadas. Para que el código sea portable entre distintos sistemas de desarrollo, se puede sustituir este layout por null.
- Otra de las ventajas que hemos encontrado utilizando esta forma de dibujar las estructuras ha sido la composición de imágenes, unas encima de otras. Hemos sido capaces de controlar la superposición de distintos elementos en la pantalla, pudiendo decidir que elemento se pone por encima y cual por debajo. Esto se consigue ya que el elemento que está en la primera posición del **Container**, es el que está por encima de todos los demás, y el que está en la última posición, es el que está por debajo de todos los elementos.

En cuanto a las imágenes que hemos utilizado, hemos elegido el formato PNG (Portable Network Graphics) ya que es un formato gráfico basado en un algoritmo de compresión sin pérdida para bitmaps no sujeto a patentes. Nos hemos decantado por este formato por varias razones:

- Este formato es reconocido por diferentes sistemas operativos como Windows, Linux y Mac.

- Nos permite almacenar imágenes de hasta 32 bits de profundidad con canal alfa. Esto quiere decir que podemos tener imágenes con transparencia, muy útiles para superponer imágenes. Para la creación de las imágenes que hemos utilizado en la herramienta, hemos usado el programa Axialis IconWorkshop 6.0.

En la **figura 153** se muestran diferentes superposiciones entre imágenes, así como la transparencia de las mismas. En este caso concreto vemos que la tubería está por encima de todos los elementos y al tener trasparencia en su interior, da la sensación de que realmente los elementos circulan por ella. Además las flechas se colocan por encima de los elementos, e incluso de la tubería.

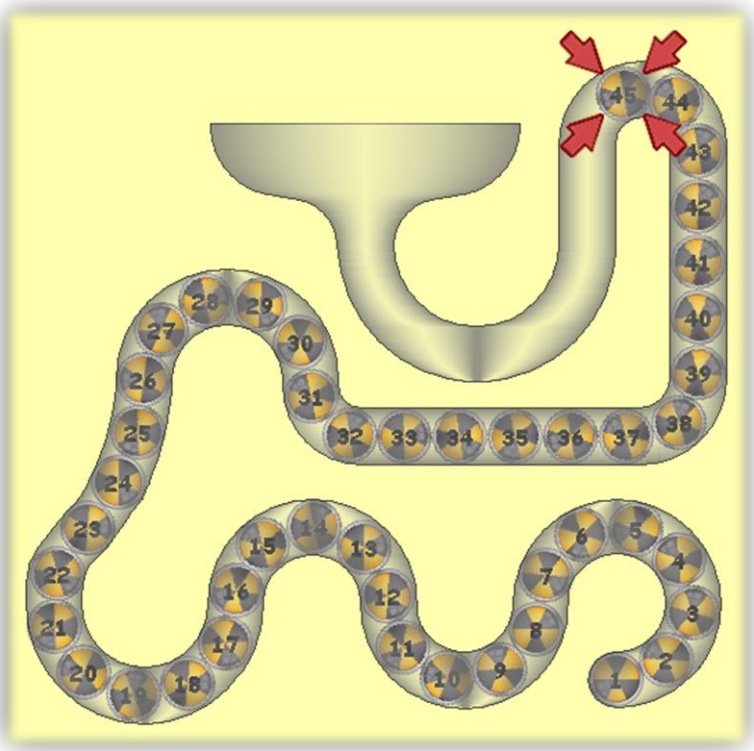


Figura 1533 – Muestra de imágenes con transparencia y superposición

6.6.- Estructuras de Datos

Independientemente de la visualización y de la estructura de datos con la que trabajemos, vamos a almacenar los elementos gráficos en arrays de una dimensión.

Concretamente, todas las clases gráficas tienen como atributo este array y se llama vectorElementos.

Este vectorElementos es muy importante ya que, aunque se podría acceder a los elementos gráficos desde el **JPanel** que los contiene, la ordenación de los mismos tanto en el vectorElementos como en el **JPanel** es distinta y significativa.

- Como ya hemos comentado en la **sección 6.5**, en el **JPanel** los elementos se ordenan en función de cuáles tienen que estar en una plano superior (los primeros) y cuáles tienen que estar en un plano inferior (los últimos). Además este **JPanel** contiene otros objetos que no son elementos de la estructura como **JLabel's** con texto, el **JLabel** con la imagen del contenedor de la estructura, etc…
- En el vectorElementos, el orden viene dado por las acciones creadas en la implementación de la estructura de datos. Las acciones de las operaciones constructoras de todas las estructuras de datos, en concreto las operaciones que añaden elementos, son las que determinan que elemento se ha de insertar, y en qué posición. Esto es porque la implementación de la estructura de datos es la que gobierna el funcionamiento de la parte gráfica.

En cuanto a la implementación de las animaciones, hay que comentar que se han realizado mediante threads o hilos. El funcionamiento básico de nuestros hilos consiste en la ejecución periódica del código de las acciones. El tiempo de espera entre una ejecución y la siguiente, dentro del mismo hilo, viene dado por la velocidad seleccionada por el usuario en la ventana de la estructura.

La secuencia de animaciones que representan una acción suelen estar divididas en tramos/pasos. La ejecución de la acción termina cuando se ha concluido el último paso. Una acción está dividida en pasos para mostrar secuencialmente lo que ocurre al animar la acción. Estos pasos, representan las instrucciones que forman una acción, así por ejemplo la acción "añade" de los árboles binarios de búsqueda, se divide en buscar la posición de inserción y en insertar el elemento. En las visualizaciones de implementación de las estructuras, los pasos de la acción los marcan las instrucciones de pseudocódigo que se muestran. Ahora bien, si en un paso, el movimiento de los objetos se puede dividir en movimientos más simples, lo dividimos en tramos.

A continuación veamos los aspectos más importantes de las implementaciones de cada una de las visualizaciones de cada una de las estructuras de datos:

6.6.1.- Pilas

La estructura de datos Pila se ha implementado usando un array para contener sus elementos, es decir, se ha elegido una implementación estática. Esto ha hecho que la equivalencia entre la implementación de la estructura de datos y el vectorElementos de la parte gráfica, sea directa, es decir, las posiciones de los elementos en el array que representa la pila, son las mismas que para sus elementosGraficos correspondientes, del vectorElementos.

6.6.1.1.- Visualización Usuario

Para representar la Pila en su visualización de usuario, hemos diseñado el circuito que se muestra en las **figuras 154**, **155** y **156**. A simple vista, parece un complejo circuito lleno de curvas, pero no es más que una trayectoria descrita a base de rectas y circunferencias.

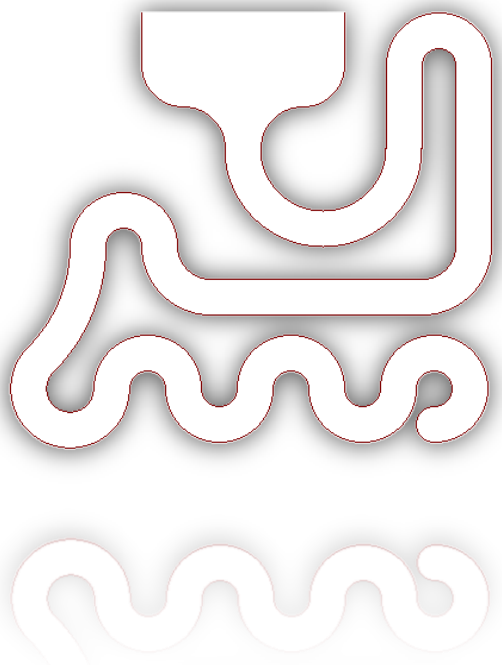


Figura 154 – Boceto de la tubería de la

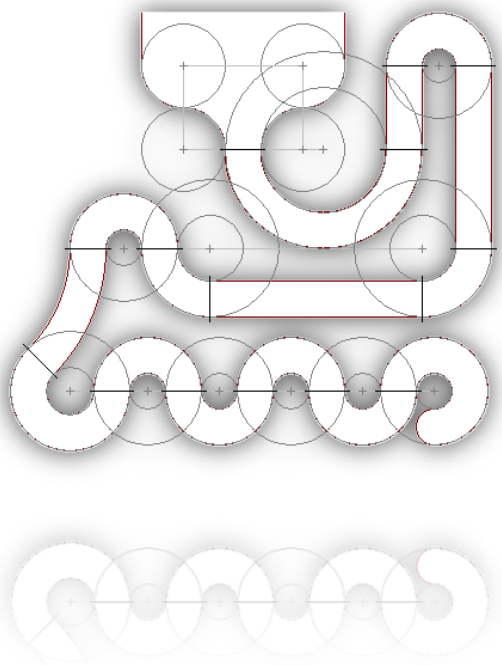


Figura 155 – Diseño de la tubería de la

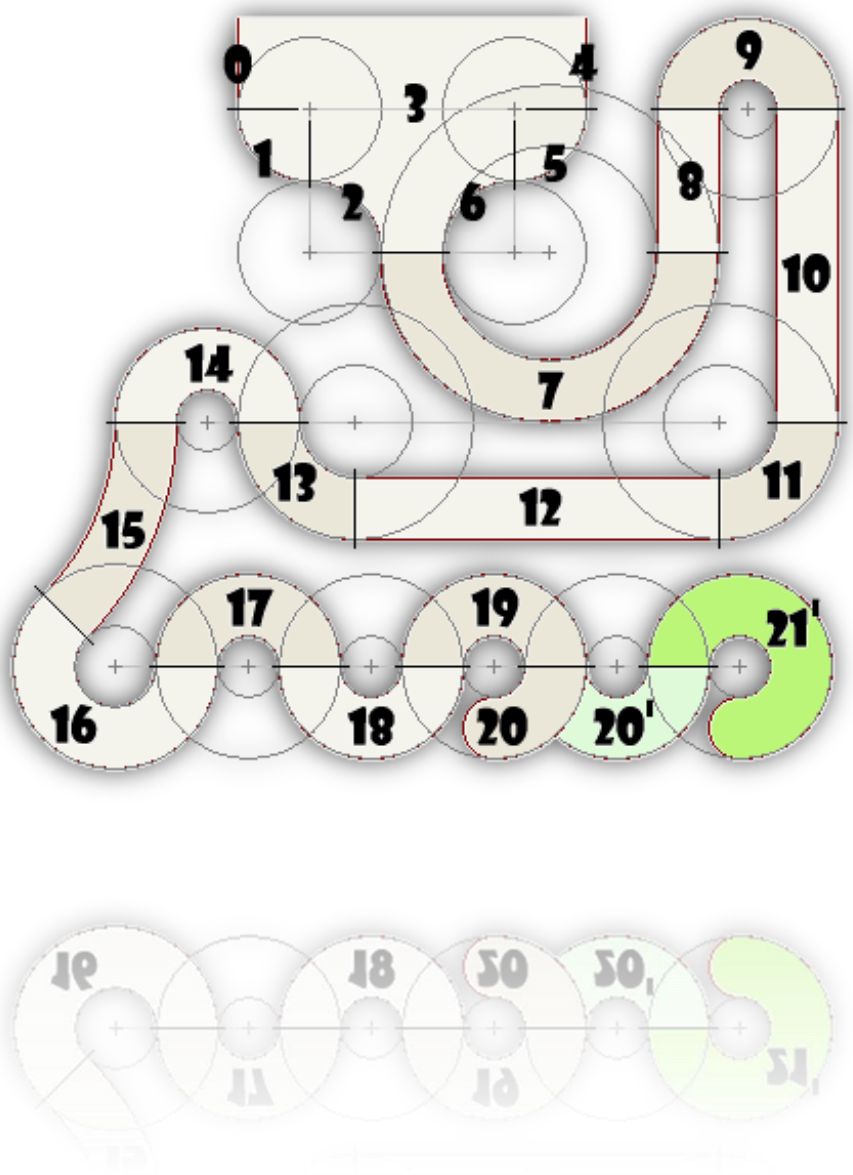


Figura 156 – Diseño final de la tubería de la Pila de Usuario

En un primer momento, se diseñó esta tubería de forma que tuviera 21 tramos. Pero al tratar el movimiento de los elementos, nos dimos cuenta que las circunferencias que se describían en la parte inferior, tenían un radio demasiado pequeño. Por eso, la eliminación de los tramos 20' y 21', y la adición del tramo 20, nos permitió aumentar el radio de esas trayectorias circulares.

Los datos concretos utilizados para describir las trayectorias de cada uno de los tramos, se muestran en la **tabla 9**. A partir de estos datos, es sencillo implementar el movimiento de un elemento a lo largo de la tubería.

Tramo	Posición Centro (x,y)	Radio	Angulo Inicial	Angulo Final	Posición Inicial (x,y)	Posición Final (x,y)	Sentido	Tipo
0	✖	✖	✖	✖	(134,-40)	(134,17)	↻	Recta
1	(174,17)	40	π/2	π	✖	✖	↻	Circular
2	(174,107)	50	3π/2	2π	✖	✖	↻	Circular
3	✖	✖	✖	✖	(224,-40)	(224,107)	↓	Recta
4	✖	✖	✖	✖	(314,-40)	(314,17)	↓	Recta
5	(274,17)	40	0	π/2	✖	✖	↻	Circular
6	(274,107)	50	3π/2	π	✖	✖	↻	Circular
7	(304,107)	80	π	0	✖	✖	↻	Circular
8	✖	✖	✖	✖	(384,107)	(384,37)	↑	Recta
9	(424,37)	40	π	2π	✖	✖	↻	Circular
10	✖	✖	✖	✖	(464,37)	(464,207)	↓	Recta
11	(424,207)	40	0	π/2	✖	✖	↻	Circular
12	✖	✖	✖	✖	(424,247)	(224,247)	←	Recta
13	(224,207)	40	π/2	π	✖	✖	↻	Circular
14	(124,207)	60	2π	π	✖	✖	↻	Circular
15	(-96,207)	160	π	π/4	✖	✖	↻	Circular
16	(64,367)	66	5π/4	0	✖	✖	↻	Circular
17	(185,367)	55	π	2π	✖	✖	↻	Circular
18	(295,367)	55	π	0	✖	✖	↻	Circular
19	(405,367)	55	π	2 π	✖	✖	↻	Circular
20	(405,367)	55	0	π/2	✖	✖	↻	Circular

Tabla 9 – Descripción de los tramos de la Pila de Usuario

Si el tramo describe una **trayectoria recta**, el movimiento consiste en aumentar o disminuir la coordenada x o la coordenada y, en función de la dirección (vertical, horizontal) y el sentido.

Si el tramo describe una **trayectoria circular**, el movimiento consiste en girar la posición del elemento respecto de un centro y un radio. Dado un elemento en una

cierta posición, queremos que gire un ángulo α a lo largo de una circunferencia de centro (x_0, y_0) y radio r. Además, es necesario que sepamos el ángulo inicial del elemento, β, de forma que al ángulo final sobre la circunferencia, después del giro α, será $t = \beta \pm \alpha$. Se sumará o se restará en función del sentido del giro, de forma que si es a favor de las agujas del reloj, se sumará, y si es en contra, se restará. La nueva posición del elemento viene dada por las siguientes ecuaciones:

$$\left.\begin{aligned} x &= x_0 + r \cdot \cos(t) \\ y &= y_0 + r \cdot \operatorname{sen}(t) \end{aligned}\right\}$$

El ángulo de giro en cada instante depende del radio. De esta forma cuanto mayor sea el radio, r, el ángulo de giro, α, tiene que ser más pequeño para conservar la velocidad lineal del elemento a lo largo del recorrido.

Independientemente del tramo, y aparte de la traslación que acabamos de explicar, la imagen del elemento rota sobre su centro. Con la combinación de los movimientos de rotación y traslación, se ha querido dar el efecto de que el elemento rueda a lo largo de toda la tubería. Hacemos notar que se ha tenido especial cuidado en la física del movimiento, es decir, el elemento rota en un sentido o en otro, y a más o menos velocidad, en función del lado sobre el que gira y la dirección del movimiento.

Hay que tener en cuenta que los elementos solo se van a mover cuando se realicen las operaciones de apilar y desapilar. Además los valores de la **tabla 9**, Posición Inicial (x,y), Posición Final (x,y), Angulo Inicial y Angulo final, son propios de las operación apilar. Para la operación desapilar los valores serán los mismos pero cambiando inicial por final y viceversa. En este caso, los elementos no tienen movimiento rotacional.

En la acción "añade", el elemento que se está añadiendo no tiene por qué recorrer todos los tramos, alcanza su posición final cuando se choca con el último elemento de la pila. Es decir, el elemento se para cuando las distancia entre los centros de éste y el último elemento de la pila es menor o igual que dos veces el radio del elemento, esto es, el tamaño del elemento.

6.6.1.2.- VISUALIZACIÓN IMPLEMENTACIÓN ESTÁTICA

En la visualización estática de las pilas se ha usado, para representar el vector que contiene los elementos, un contenedor con capacidad para 52 elementos que los distribuye en 4 filas de 13 posiciones cada una (ver **figura 157**).

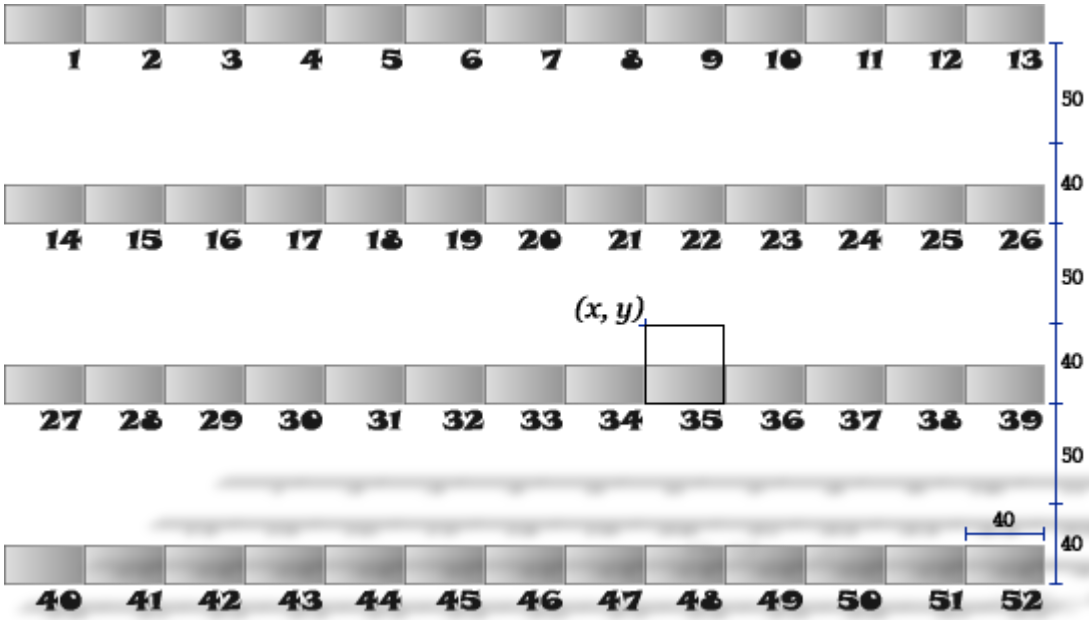


Figura 157 – Contenedor de la Pila Estática

Como se ha comentado anteriormente (ver **sección 6.4**), las acciones de las pilas que se pasan a la parte gráfica de la aplicación, contienen el elemento sobre el que se aplica la acción y su posición en el vector de elementos. En todas las acciones que ejecuta la parte grafica, menos en "añade", la posición del elemento es relevante ya que gracias a ella podemos acceder al elemento y así a su posición en el panel.

En el caso de la acción añade tenemos que trasformar la posición del vector a unas coordenadas cartesianas para colocar el elemento en el panel de animación ((x,y) en la **figura 157**). Así:

$$\left.\begin{aligned} x &= tamElemento \cdot columna + traslacionX \\ y &= (tamElemento + 50) \cdot fila + traslacionY \end{aligned}\right\} \quad donde \quad \begin{aligned} columna &= posicion \ \mathbf{mod}\ 13 \\ fila &= posicion \ \mathbf{div}\ 13 \\ posicion &\in [0..51] \\ tamElemento &= 40 \end{aligned}$$

Las variables traslacionX y traslacionY se utilizan para centrar la estructura en el panel, y dependen del tamaño de este.

Todos los movimientos que describen los elementos son verticales, luego su implementación es muy sencilla, ya que solo habrá que modificar la coordenada y de la posición en el panel.

6.6.1.3.- Visualización Implementación Dinámica

En la visualización dinámica de las pilas, por supuesto, no existe ningún contenedor que encierre a los elementos, pero estos deben ocupar unas posiciones determinadas para que en la pantalla se puedan mostrar los 52 elementos, que como máximo pueden almacenar nuestras pilas.

El primer elemento está colocado en la esquina inferior derecha, y a partir de éste, el resto de los elementos se colocan como se muestra en la **figura 158**

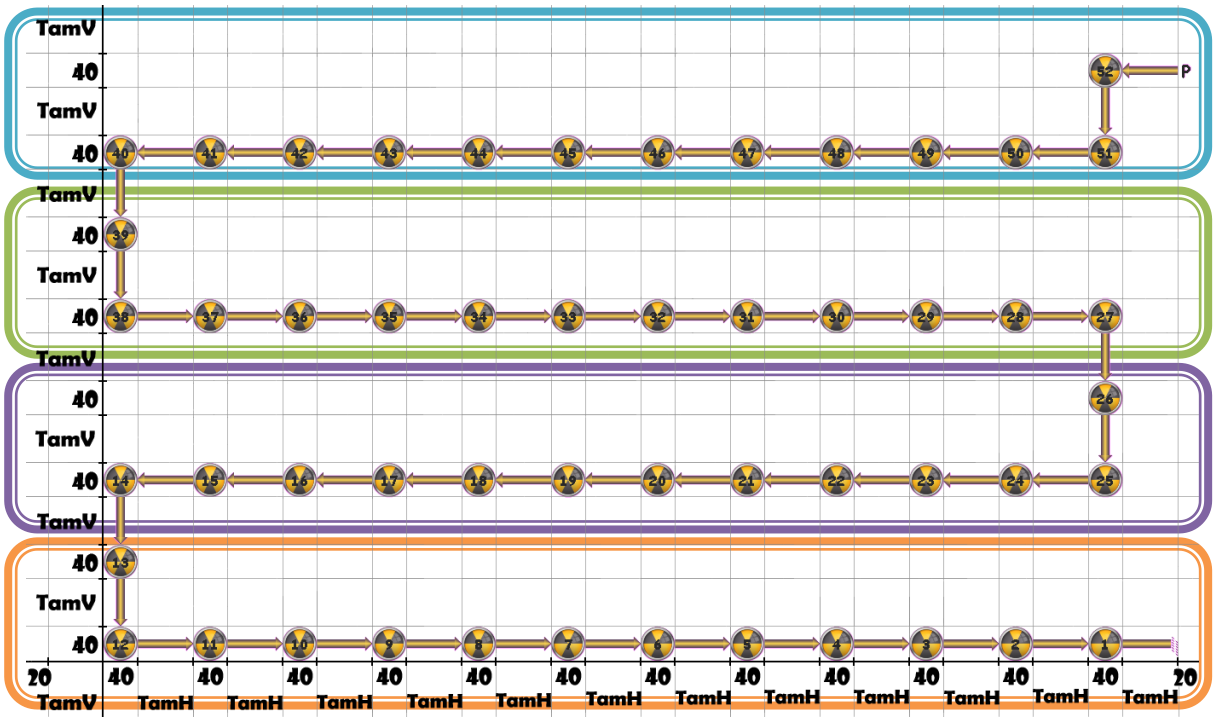


Figura 158 – Disposición de los elementos en la Pila Dinámica

El tamaño de las flechas depende del tamaño del panelAnimacion. Así, el tamaño de las flechas horizontales y el tamaño de las flechas verticales viene dado por:

$$\left.\begin{aligned} TamH &= \frac{AnchoPanelAnimacion - 13 \cdot tamElemento}{13} \\ TamV &= \frac{AltoPanelAnimacion - 8 \cdot tamElemento}{9} \end{aligned}\right\} donde\ tamElemento = 40$$

Como en el caso de las pilas estáticas, tenemos que poder trasformar un índice del vector de elementos a una posición en el panel. Para ello:

$$\left.\begin{aligned} fila &= indice\ \boldsymbol{div}\ 13 \\ columna &= indice\ \boldsymbol{mod}\ 13 \end{aligned}\right\}\ donde\ indice \in [0..51]$$

Veamos en la **tabla 10** las ecuaciones para calcular la posición (x,y) del elemento en función de los valores de fila y columna:

FILA ES PAR COLUMNA ≠ 12	$x = \frac{tamElemento}{2} + (12 - columna) \cdot TamH + (11 - columna) \cdot tamElemento$
	$y = (8 - 2 \cdot fila) \cdot TamV + (7 - 2 \cdot fila) \cdot tamElemento$
FILA ES PAR COLUMNA = 12	$x = x \cdot (columna - 1) = \frac{tamElemento}{2} + TamH$
	$y = (7 - 2 \cdot fila) \cdot TamV + (6 - 2 \cdot fila) \cdot tamElemento$
FILA ES IMPAR COLUMNA ≠ 12	$x = \frac{tamElemento}{2} + (columna + 1) \cdot TamH + (columna) \cdot tamElemento$
	$y = (8 - 2 \cdot fila) \cdot TamV + (7 - 2 \cdot fila) \cdot tamElemento$
FILA ES IMPAR COLUMNA = 12	$x = x \cdot (columna - 1) = \frac{tamElemento}{2} + 11 \cdot TamH$
	$y = (7 - 2 \cdot fila) \cdot TamV + (6 - 2 \cdot fila) \cdot tamElemento$

Tabla 10 – Ecuaciones para de posición del objeto en función de su zona

En la implementación dinámica no se produce ningún tipo de traslación sobre los elementos, salvo en la acción elimina, siendo está horizontal o vertical. Para realizar las animaciones, a las flechas se les ha dotado de la capacidad de aumentar y disminuir su tamaño.

6.6.2.- Colas

La estructura de datos Cola se ha implementado usando un array circular para contener sus elementos, es decir, se ha elegido una implementación estática. Esto ha hecho que la equivalencia entre la implementación de la estructura de datos y el vectorElementos de la parte gráfica, sea directa, es decir, las posiciones de los elementos en el array que implementa la cola, son las mismas que para sus elementosGraficos correspondientes, del vectorElementos.

6.6.2.1.- Visualización Usuario

Para representar la Cola en su visualización de usuario, diseñamos el circuito que se muestra en la **figura 159**.

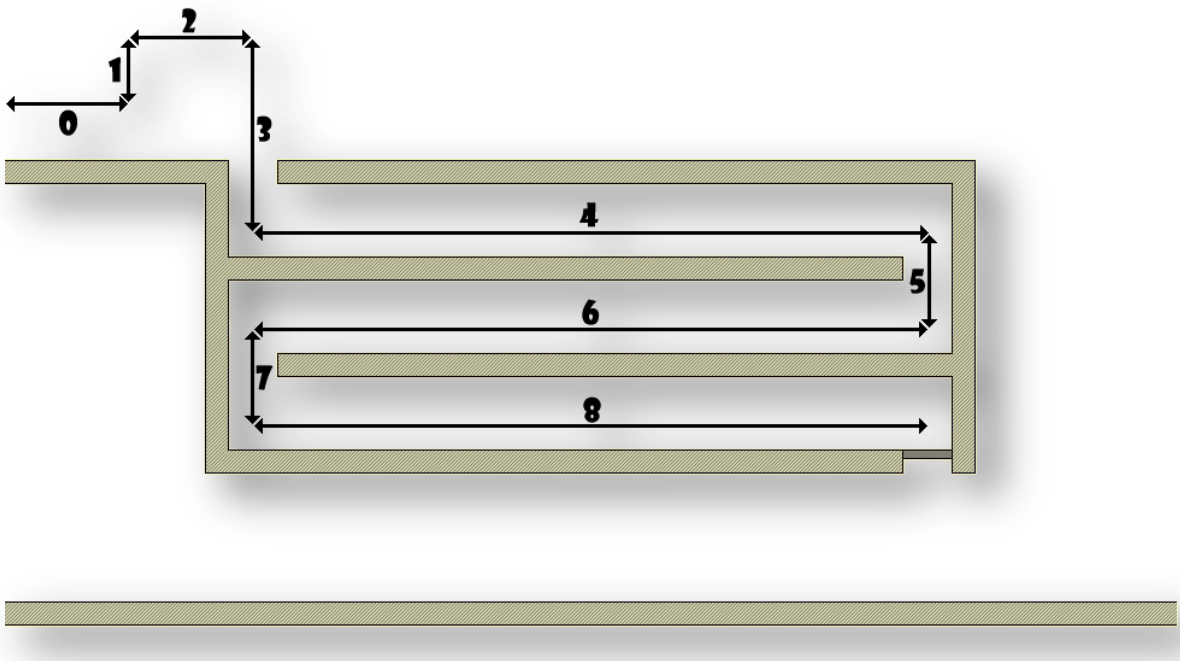


Figura 159 – Diseño del circuito de la Cola de Usuario

Como en el caso de las pilas de usuario, el circuito está dividido en tramos. En la **figura 159** se muestran los tramos que describen los movimientos que tiene que hacer el elemento al ejecutar la acción añade. Como se observa, la complejidad de los movimientos es mucho menor que en la pila de usuario, al ser horizontales y verticales.

Al igual que en la pila de usuario, se ha implementado el movimiento rotacional de los elementos, para dar la impresión de que ruedan a lo largo de la estructura.

Para la acción "elimina" hemos considerado los siguientes casos en función de la distancia de la posición del elemento al lugar donde le recogerá el camión (distancia1) y la distancia de la posición del camión hasta el lugar de recogida (distancia2):

- Si distancia1 > distancia2: solo avanza el elemento.
- Si distancia1 < distancia2: solo avanza el camión.
- Si distancia1 = distancia2: avanzan los dos a la vez hasta que el camión recoge al elemento.

A partir de la recogida, el elemento y el camión avanzan juntos hasta desaparecer del panel.

6.6.2.2.- Visualización Implementación Estática

En la visualización estática de las colas se ha usado, para representar el array circular con el que se implementa las colas desde el punto de vista estático, un contenedor con capacidad para 40 elementos que los distribuye en forma circular (ver **figura 160**).

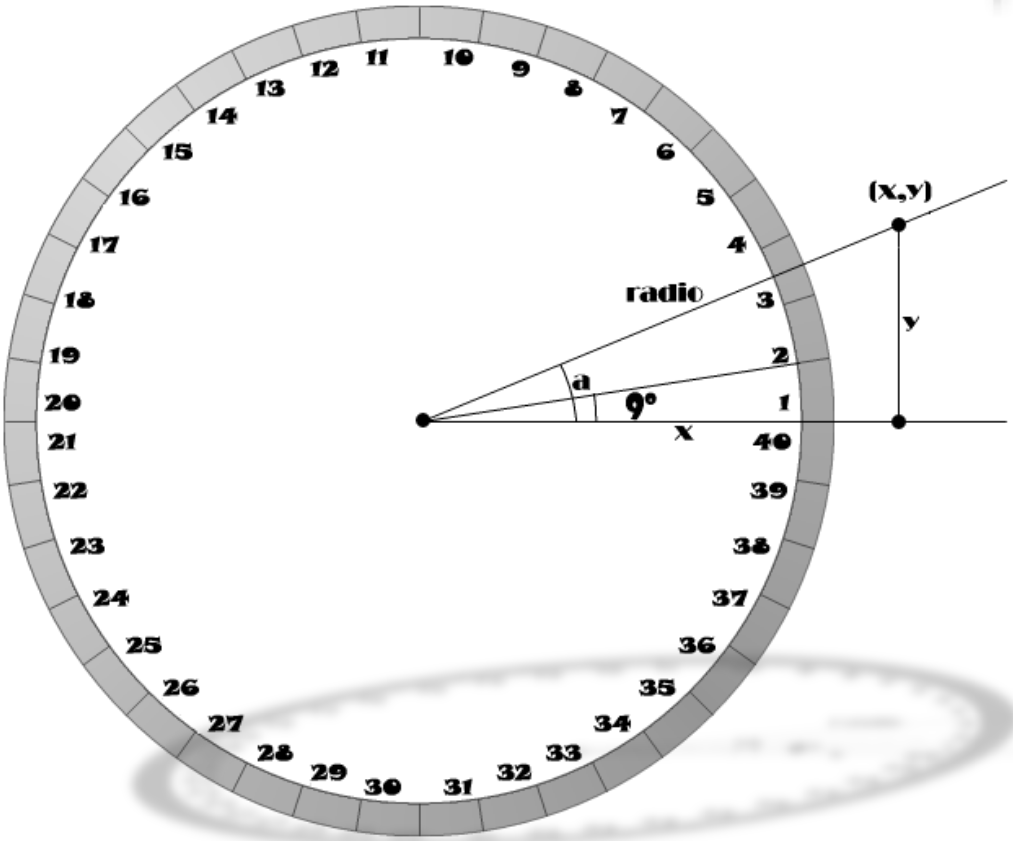


Figura 160 – Contenedor de la Cola Estática

El movimiento que realizan los elementos es radial. Esto es, las trayectorias que siguen los elementos, son rectas que pasan por el centro de la circunferencia que describe el contenedor de la cola. Hemos utilizado un poco de trigonometría para calcular, en cada instante, la posición (x,y) del elemento:

$$cos(a) = \frac{x}{radio} \rightarrow x = radio \cdot cos(a)$$

$$sen(a) = \frac{y}{radio} \rightarrow y = radio \cdot sen(a)$$

Por lo tanto necesitamos saber el ángulo a, que forma la horizontal con la recta que describe la trayectoria del movimiento. Como hemos marcado que el número máximo de elementos es 40, para saber qué arco abarca cada una de las posiciones gráficas del vector, hacemos el siguiente cálculo:

$$\frac{360°}{40} = 9°$$

Sabiendo esto, si tenemos el indice del elemento dentro del vector, podemos calcular de forma sencilla, las posiciones x e y que le corresponden:

$$a = -(4{,}5° + 9° \cdot indice) \cdot \frac{\pi}{180°}$$

$$x = radio \cdot cos\left(-(4{,}5° + 9° \cdot indice) \cdot \frac{\pi}{180°}\right)$$

$$y = radio \cdot sen\left(-(4{,}5° + 9° \cdot indice) \cdot \frac{\pi}{180°}\right)$$

donde radio es la distancia desde el centro de la circunferencia hasta el centro del elemento.

Para describir los movimientos de las animaciones de las acciones añade y elimina, disminuimos y aumentamos el radio, respectivamente. En la acción de añade el movimiento termina cuando el radio es igual al radio de la circunferencia. En la acción de elimina el movimiento termina cuando el radio supera los límites del panelAnimacion.

Esta posición (x,y) que acabamos de calcular, es una posición relativa al centro de la circunferencia. Para transformar estas coordenadas con referencia al panelAnimacion, hacemos:

$$x = x + radio + traslacionX - \frac{tamElemento}{2}$$

$$y = y + radio + traslacionY - \frac{tamElemento}{2}$$

donde las variables traslacionX y traslacionY se utilizan para centrar el dibujo en el panel, dependen del tamaño de este.

6.6.2.3.- Visualización Implementación Dinámica

Al igual que en la visualización de implementación dinámica de las pilas, en la visualización dinámica de las colas, no existe ningún contenedor que encierre a los

elementos, pero estos deben ocupar unas posiciones determinadas para que en la pantalla se puedan mostrar los 40 elementos, que como máximo pueden almacenar nuestras colas.

El primer elemento empieza a colocarse en la esquina superior izquierda, y a partir de éste, el resto de los elementos se colocan como se muestra en la **figura 161**.

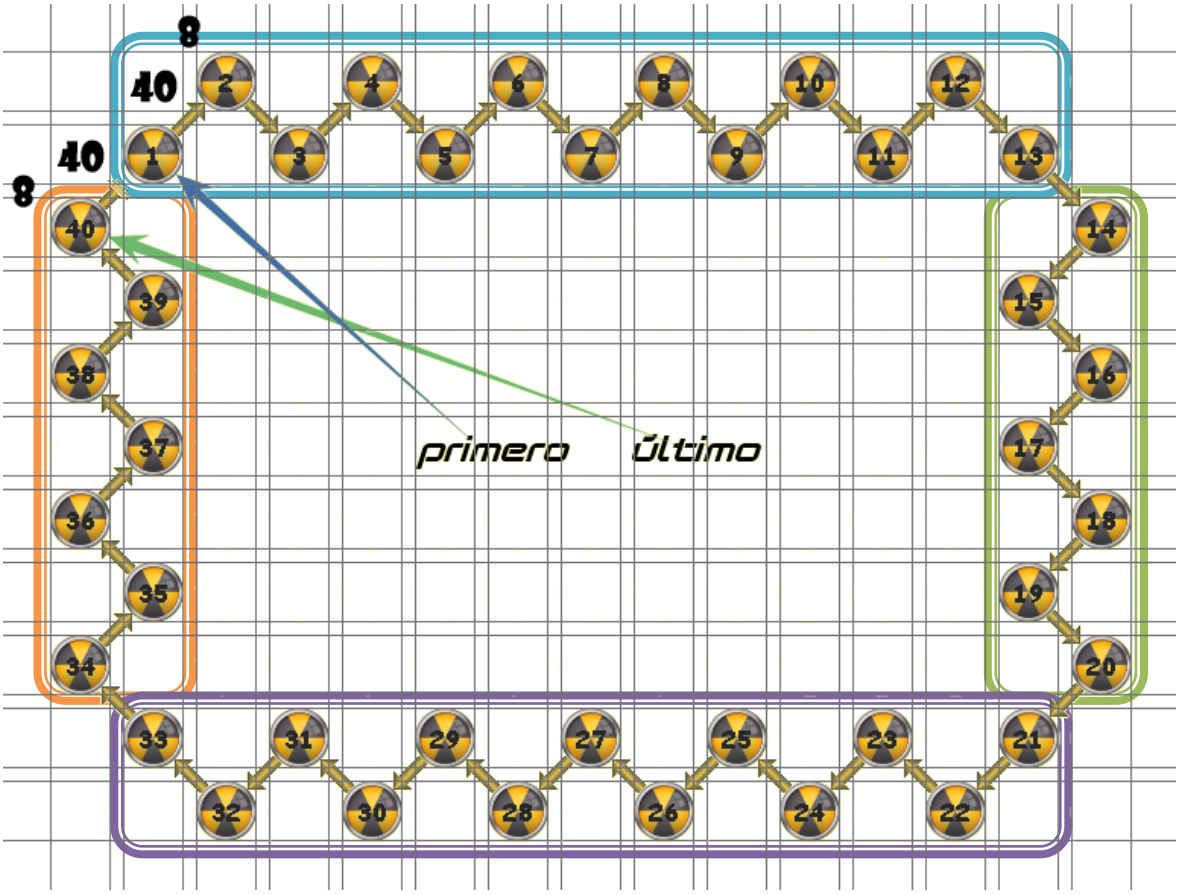


Figura 161 – Disposición de los elementos en la Cola Dinámica

Una de las diferencias con la implementación dinámica de las pilas, es que el tamaño de las flechas en la cola es fijo. Para que los 40 elementos, entraran en una pantalla de por lo menos una resolución de 1024x768, la distancia entre los elementos debía ser de 8 pixels (huecoElementos)

Como en casos anteriores, tenemos que poder trasformar un índice del vector de elementos a una posición en el panel. Para ello dividimos el panel en las cuatro zonas indicadas en la **figura 161**. Para ello veamos la **tabla 11** que contiene las ecuaciones de posición que hay que utilizar en cada caso:

$0 \leq INDICE \leq 12$	$x = (indice + 1) \cdot (tamElemento + huecoElementos) + traslacionX$	
	SI INDICE ES PAR	$y = tamElemento + huecoElementos + traslacionY$
	SI INDICE ES IMPAR	$y = traslacionY$
$13 \leq INDICE \leq 19$	$\boldsymbol{indiceAux = indice - 13}$ **entonces** $\boldsymbol{indiceAux} \in [\mathbf{0}..\mathbf{6}]$	
	$x = 13 \cdot (tamElemento + huecoElementos) + traslacionX$	
	$y = (indiceAux + 2) \cdot (tamElemento + huecoElementos) + traslacionY$	
	SI INDICE ES IMPAR	$x = x + tamElemento + huecoElementos$
$20 \leq INDICE \leq 32$	$\boldsymbol{indiceAux = indice - 20}$ *entonces* $\boldsymbol{indiceAux} \in [\mathbf{0}..\mathbf{12}]$	
	$x = (13 - indiceAux) \cdot (tamElemento + huecoElementos) + traslacionX$	
	$y = 9 \cdot (tamElemento + huecoElementos) + traslacionY$	
	SI INDICE ES IMPAR	$y = y + tamElemento + huecoElementos$
$33 \leq INDICE \leq 39$	$\boldsymbol{indiceAux = indice - 33}$ **entonces** $\boldsymbol{indiceAux} \in [\mathbf{0}..\mathbf{6}]$	
	SI INDICE ES PAR	$x = tamElemento + huecoElementos + traslacionX$
	SI INDICE ES IMPAR	$x = traslacionX$
	$y = (8 - indiceAux) \cdot (tamElemento + huecoElementos) + traslacionY$	

Tabla 11 - Ecuaciones para de posición del objeto en función de su zona

Las variables traslacionX y traslacionY se utilizan para centrar la estructura en el panel, y dependen del tamaño de este.

En la implementación dinámica no se produce ningún tipo de traslación sobre los elementos. Para realizar las animaciones, a las flechas y a los elementos se les ha dotado de la capacidad de rotar sobre sí mismos y de variar su tamaño.

6.6.3.- Arboles Binarios de Búsqueda

La estructura de datos árbol binario de búsqueda se ha implementado usando punteros, que relacionan la raíz con sus hijos, es decir, se ha elegido una implementación dinámica.

Cada accionArbolBinarioBusqueda (ver **sección 6.4.**) tiene un vector `caminoAccion`, que contiene los índices de los elementos que están involucrados en la acción. Estos índices indican la posición que ocupa el nodo si recorremos el árbol en anchura, esto quiere decir que si un elemento tiene un índice i, su hijo izquierdo tendría el índice $2 \cdot i + 1$ y su hijo derecho el índice $2 \cdot i + 2$. Para realizar la traducción de estos índices a los elementos gráficos correspondientes, en la parte gráfica de la aplicación, existe un array de posiciones, VectorPosiciones, donde el contenido del índice es la posición del VectorElementos donde está el elemento gráfico.

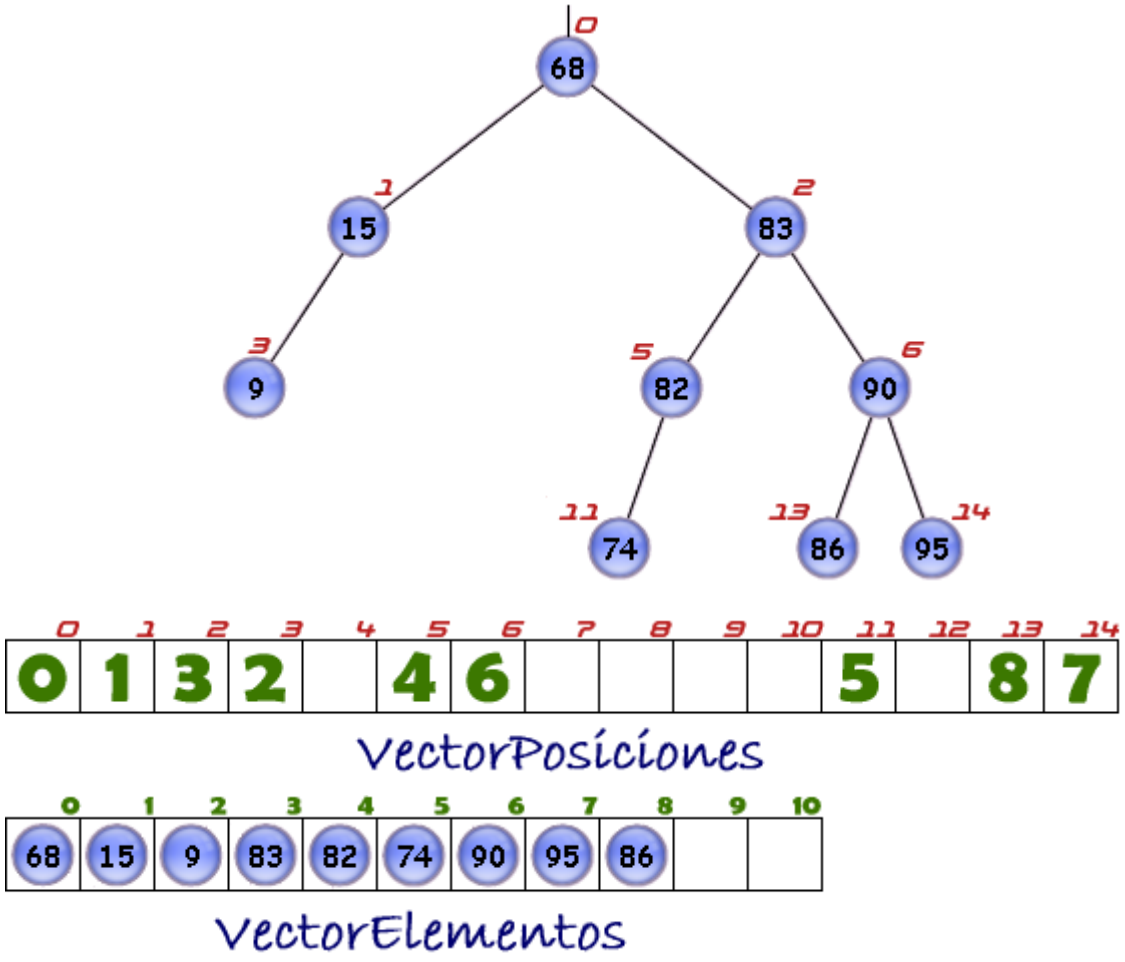


Figura 162 – Ejemplo de la relación del vector posiciones y del vector elementos

En la **figura 162** se muestra un ejemplo de un árbol binario de búsqueda, donde se aprecia la relación existente entre el VectorPosiciones y el VectorElementos.

Si el árbol no es completo significa que el array de posiciones, de **ArbolBinarioBusquedaGrafico**, tendrá huecos entre las posiciones ocupadas. A pesar de esto, el vectorElementos no tiene porque tener huecos ya que cuando se añade un elemento se añade de forma secuencial. Para una mayor eficiencia del espacio del vectorElementos, existe un vector de huecos que se actualiza cuando se elimina un elemento y se usa para añadir elementos en los espacios libres.

6.6.3.1.- Visualización Usuario

La implementación de los árboles binarios de búsqueda (extensible a otro tipo de árboles), se basa en que los elementos gráficos que representan cada uno de los nodos del árbol, dibujan automáticamente las rectas que los unen con sus progenitores (de esto se encarga **NexoPadreHijo**). Al ser los propios elementos los que dibujan estas rectas, en función de su posición y la de su padre, únicamente nos hemos tenido que preocupar en asignar a cada nodo su padre, e implementar sus movimientos, ya que con esta información, (posición padre y posición del hijo) dibujar la recta se hace de forma automática independientemente de cómo estén colocados. Esta sofisticada forma de tratar los elementos del árbol, ha hecho que sea posible tanto el redimensionamiento como la recolocación del mismo (ver **sección 6.3**).

Hasta este momento, en todos los movimientos que habíamos implementado, se variaba un único parámetro para calcular la nueva posición del objeto:

- **Movimientos circulares** (pila usuario), donde aumentábamos o disminuíamos un ángulo.
- **Movimientos radiales** (cola estática), donde aumentábamos o disminuíamos un radio.
- **Movimientos horizontales**, donde aumentábamos o disminuíamos la coordenada x.
- **Movimientos verticales**, donde aumentábamos o disminuíamos la coordenada y.

Pero ninguno de estos movimientos nos servía para los árboles. Por ello hemos creado otro tipo, que son los movimientos de punto a punto. En este tipo de movimientos se trata de mover los objetos desde un punto inicial hasta un punto final. Como en este tipo de movimientos, partimos de una posición inicial (x_1,y_1) y queremos llegar a otra posición (x_2,y_2), se tienen que variar dos parámetros.

Una primera alternativa fue variar uno de los parámetros de forma fija, mientras que la variación del otro se calculaba por la pendiente de la recta. Pero esto ocasionaba que en trayectorias de distintas inclinaciones parecía que el objeto se movía a distinta velocidad. Por ejemplo, si fijábamos la x a 2, el valor del incremento de la y será muy distinto en la trayectoria de una recta que forme 45º con la horizontal, que en una que forme 88º. En los dos casos el valor de la x es el mismo, pero, en el primero el valor de la y es 2 y en el segundo es, aproximadamente, 57.

Para conseguir un movimiento coherente, independientemente de la inclinación de la recta que sigue la trayectoria de los objetos, ideamos lo siguiente.

Calculamos la distancia entre los puntos inicial (x_1,y_1) y final (x_2,y_2):

$$\left.\begin{array}{l} variacionX = x_2 - x_1 \\ variacionY = y_2 - y_1 \end{array}\right\} \text{ por lo que } distancia = \sqrt[2]{variacionX^2 + variacionY^2}$$

Dividimos esta distancia entre 2 de forma que queremos que el objeto se mueva de 2 pixels en 2 pixels. El resultado de esta división entera, va a ser el número de veces que vamos a variar las coordenadas del objeto para alcanzar la posición final.

$$numTramos = \left\lfloor \frac{distancia}{2} \right\rfloor$$

Calculamos los incrementos de las variables x e y, correspondientes a la variación de 2 pixels sobre la recta.

$$incrementoX = \left\lfloor \frac{variacionX}{numTramos} \right\rfloor$$

$$incrementoY = \left\lfloor \frac{variacionY}{numTramos} \right\rfloor$$

Estos incrementos se redondean ya que las coordenadas de los elementos tienen que ser valores enteros. Esto produce una pérdida de precisión que provoca que en el último tramo pueda que no se llegue exactamente a la posición final (x_2,y_2), por lo que nos aseguramos de que llegue a la posición final colocando el objeto en esa posición.

En nuestro afán por complicarnos la vida, hemos hecho que los objetos tengan aceleración y deceleración en sus movimientos. Esto se ha conseguido modificando el retardo (ratio, tiempo de espera del hilo) en cada posición del movimiento. La variación del retardo viene dado por la gráfica de la **figura 163**:

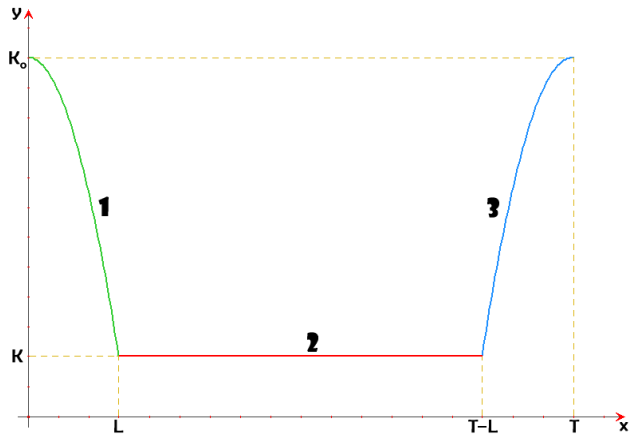


Figura 163 – Gráfica que representa la variación del ratio en función de la posición del objeto

El significado de cada una de las variables es el siguiente:

- **y** es el tiempo de espera en milisegundos para pasar de una posición a la siguiente, según la trayectoria. Es el retardo del hilo de la animación.
- **x** es la cantidad de bipixels de un movimiento. Se define un bipixel como 2 pixels.
- **T** es el número de bipixels del movimiento, es decir, numTramos.
- **L** es el límite, en bipixels, hasta donde se aplica la aceleración.
- **T-L** es el límite, en bixpixels, desde donde se aplica la deceleración.
- **K** es el retardo (ratio) elegido por el usuario para el movimiento.
- **D** es la diferencia entre el ratio real y el ratio inicial.
- $\mathbf{K_0}$ es el retardo (ratio) inicial y final del movimiento. Se calcula como **K+D**

Las ecuaciones que definen la aceleración, la velocidad constante y la deceleración del movimiento en cada tramo son, respectivamente:

1. $y = \frac{-D}{L^2}x^2 + K_0$ $\quad si\ x \leq L$

2. $y = K$ $\quad si\ L < x < T - L$

3. $y = \frac{-D}{L^2}(x - T)^2 + K_0$ $\quad si\ x \geq T - L$

7.- Cómo ampliar VEDYA

Como ya hemos comentado, la aplicación está especialmente diseñada para que las posibles ampliaciones sobre la misma sean lo más sencillas posible. Como ejemplo, supongamos que queremos añadir las colas de prioridad a la aplicación. A continuación explicaremos con detalle los pasos a seguir y finalizaremos con la ampliación del diagrama de clases representado en la **figura 164**.

7.1.- Paquete Ventanas

En la clase **VentanaMain**, se tiene que crear el enlace con la nueva estructura. Para ello modificamos lo siguiente:

- Añadimos el atributo estático `ventanaColaPrioridad` de tipo **VentanaColaPrioridad** que crearemos más tarde.
- Añadimos un nuevo **JButton** correspondiente a la nueva estructura, en este caso, el botón correspondiente a las colas de prioridad. Modificar los métodos
  - `public void animacionEntradaBotonesEstructurasDatos(..)`
  - `public void animacionSalidaBotonesEstructurasDatos(..)`

  añadiendo al vector el nuevo **JButton**. Crear el método
  - `void botonColaPrioridad_actionPerformed(..)`

  que tiene que crear la **VentanaColaPrioridad**.
- Hay que modificar los métodos
  - `void salir(..)`
  - `public void guardarSaliendo(..)`
  - `public void setAplicacionAccesible(..)`

  con la parte correspondiente a la nueva estructura de datos.

Dentro del subpaquete ventanasEstructuras, creamos la clase **VentanaColaPrioridad** que hereda de **GeneradorVentana**. Lo más sencillo, es crearla como una modificación de una clase de este tipo ya creada. En este caso, se crearía a partir de la clase **VentanaCola**. Las modificaciones que se tienen que realizar sobre esta clase son:

- Cambiar el nombre de cada uno de los **JButton**'s y **JMenuItem**'s que representan cada una de las operaciones de la estructura antigua, con los nombres de las operaciones de la estructura nueva. Hay que modificar convenientemente, si hace falta, cada uno de los métodos a los que llaman estos botones. Se pueden eliminar o añadir operaciones si es necesario.
- Cambiar el nombre de cada uno de los **JMenuItem**'s y **JRadioButtonMenuItem**'s que representan cada una de las visualizaciones de la estructura antigua, con las visualizaciones de la estructura nueva. Hay que modificar convenientemente, si hace falta, cada uno de los métodos a los que llaman estos botones. Se pueden eliminar o añadir visualizaciones si es necesario.
- Hay que modificar el método
  - `protected void abrir(..)`

  para que pueda cargar desde un archivo las operaciones que correspondan a la nueva estructura de datos.

En el subpaquete pestañas, hay que crear la clase **PestañaColaPrioridad** que hereda de **GeneradorPestaña**. Lo más sencillo, vuelve a ser, crearla como una modificación de una clase de este tipo ya creada. En este caso, se crearía a partir de la clase **PestañaCola**. Las modificaciones que se tienen que realizar sobre esta clase son:

- Cambiar el nombre de cada uno de las variables booleanas que indican si un botón operación está pulsado o no, con los nombres de las operaciones de la estructura nueva.
- Cambiar el nombre de cada uno de las estructuras gráficas que tiene la estructura de datos. De esta forma tiene que quedar como atributo `ColaPrioridadGrafica`, y suponiendo que vamos a mostrar visualizaciones de usuario, de implementación estática y de implementación dinámica, también tendremos como atributos `ColaPrioridadGraficaUsuario`, `ColaPrioridadGraficaEstatica` y `ColaPrioridadGraficaDinamica`.
- Cambiar el atributo que contiene la implementación de la estructura, quedándonos con el atributo `ColaPrioridad`.
- Modificar, consecuentemente con las nuevas operaciones de la estructura, los métodos

- `public void ejecutarOperaciones(..)`
- `public void leerOperacionSiguiente(..)`

- Modificar, si es necesario, los métodos que ejecutan cada una de las operaciones de la estructura.

7.2.- Paquete Implementación

Dentro del subpaquete interfaces, hay que crear la clase **ColaPrioridad**. Hay que tener en cuenta que:

- Esta clase es una interfaz.
- Tienen que estar las operaciones típicas que tiene una **ColaPrioridad**.

Directamente, dentro del paquete implementacion, creamos la clase **ImplementacionColaPrioridad**, teniendo en cuenta que:

- En esta clase se tiene que implementar la estructura de datos con todas las operaciones que se han puesto en el interfaz que implementa.
- Cada vez que se realiza una operación, se tiene que recoger en el vector de acciones, la información necesaria para poder dibujar las animaciones que se quieran mostrar. El vector de acciones contendrá objetos del tipo **AccionColaPrioridad**.

7.3.- Paquete Utilidades

Creamos directamente en el paquete utilidades, la clase **AccionColaPrioridad**. Hay que tener en cuenta que:

- Los atributos que tenga esta clase dependerán de las animaciones que se quieren realizar.

7.4.- Paquete Gráficos

Lo que queda por añadir es lo más complicado puesto que es la parte que tiene que dibujar y animar la estructura de datos, y esto depende totalmente de cómo lo piense el programador. De todas maneras, indicamos qué clases se tienen que crear:

Se crea en el paquete graficos, la clase **ColaPrioridadGrafica**. Lo más sencillo, es crearla como una modificación de una clase de este tipo ya creada. En este caso, se crearía a partir de la clase **ColaGrafica**. Las modificaciones que se tienen que realizar sobre esta clase son:

- Modificar el método
  - `public void ejecutarAcciones(..)`

  consecuentemente con las acciones creadas para representar cada una de las operaciones de la nueva estructura.

Como hemos supuesto que las colas de prioridad van a tener las visualizaciones de usuario, de implementación estática y de implementación dinámica, hacemos lo siguiente:

Dentro del subpaquete usuario, creamos la clase **ColaPrioridadGraficaUsuario**, a la vez que creamos en el subpaquete estatica y dinamica, las clases **ColaPrioridadGraficaEstatica** y **ColaPrioridadGraficaDinamica**, respectivamente. Lo más sencillo, es crearlas como modificaciones de clases de este tipo ya creadas. En este caso, se crearían a partir de las clases **ColaGraficaUsuario**, **ColaGraficaEstatica** y **ColaGraficaDinamica**, respectivamente. Hay que tener en cuenta que:

- Hay que cambiar el método
  - `public void dibujaColaPrioridad(..)`

  para que dibuje la estructura de datos como hayamos diseñado.
- Estas clases tienen que contener cada una, un método del tipo
  - `public void ejecutarAccion*****(..)`

  por cada acción que se haya definido. En estos métodos se implementará el código necesario para dibujar los elementos que van a intervenir en la animación. Cada uno de estos métodos ha de tener una llamada al hilo correspondiente que ejecute la animación.

7.5.- Paquete Animación

En el subpaquete usuario, creamos la clase **HiloColaPrioridadUsuario**, a la vez que creamos en los subpaquetes estatica y dinamica, las clases

HiloColaPrioridadEstatica y **HiloColaPrioridadDinamica**. Lo más sencillo, es crearlas como modificaciones de clases de este tipo ya creadas. En este caso, se crearían a partir de la clase **HiloColaUsuario**, **HiloColaEstatica** e **HiloColaDinamica**, respectivamente, teniendo en cuenta que:

- Estas clases implementan **Runnable**, es decir, son hilos que han de tener implementado el método
  - `public void run(..)`
- Para controlar la terminación de la ejecución del hilo, hemos creado el método
  - `public void finish(..)`

  Estos métodos se tendrán que modificar en función de las nuevas animaciones creadas.
- Se tendrá que modificar el resto de la clase en función de las animaciones que se quieran crear. Para ello se aconseja crear un método que ejecute cada acción.

7.6.- Comentarios a la Ampliación

Aunque no ha dado tiempo a desarrollarlo en el presente proyecto, si se ha hablado sobre cómo se integrarían dentro de la aplicación otras estructuras como árboles AVL y árboles Rojinegros.

Al ser estas estructuras, otras implementaciones alternativas de los árboles binarios de búsqueda, se ha pensado que la mejor forma de añadir estas estructuras sería, no creando una ventana particular para cada una de ellas, sino creando pestañas para cada una de estas implementaciones, haciendo que fuesen accesibles todas, ABB, árboles AVL y árboles Rojinegros, desde la misma ventana VentanaArbolBinarioBusqueda. Para ello se tendría que añadir a la ventana las opciones necesarias para que al añadir una nueva pestaña, se nos dé la posibilidad de crear una pestaña de un tipo u otro.

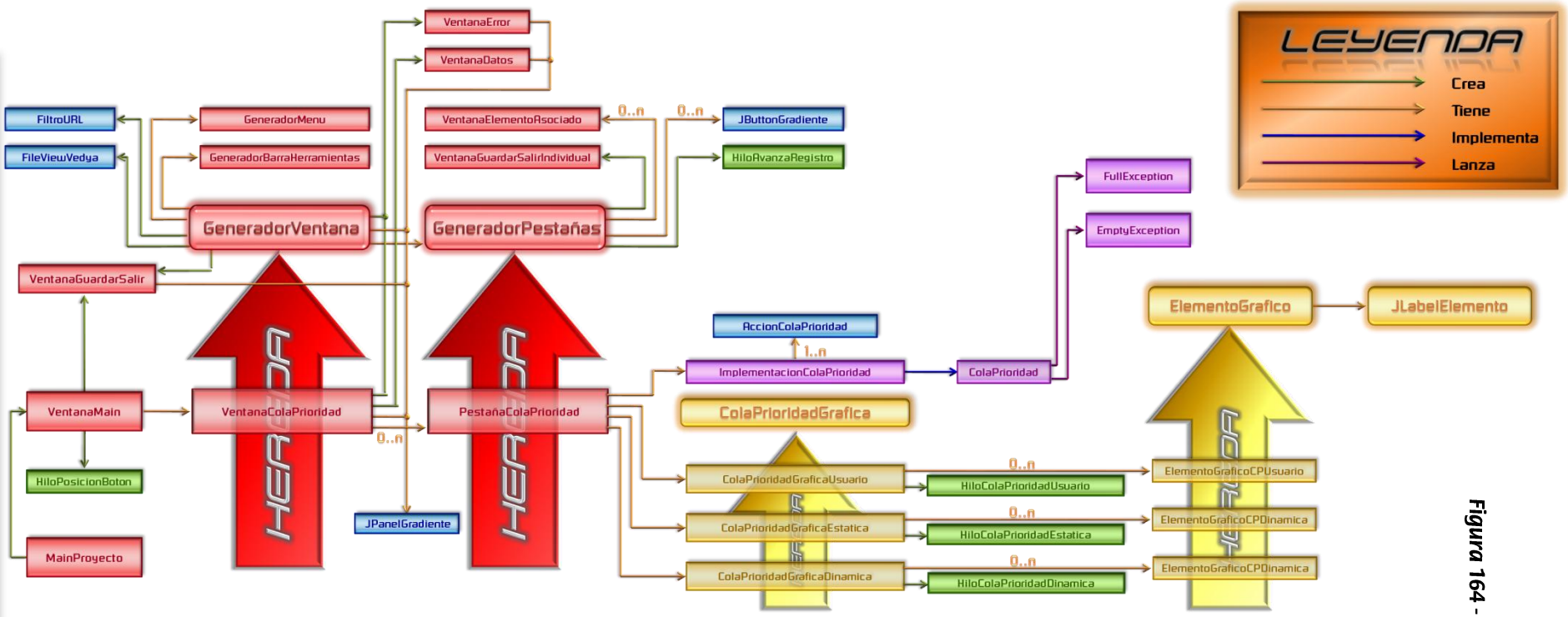


Figura 164 - Ampliación al diagrama de clases de VEDYA.

Todas las clases que se han añadido para crear las visualizaciones de la nueva estructura de datos, en este caso las colas de prioridad, siguen el esquema mencionado en la sección 4.1.

Como se puede comprobar, la interrelación que mantienen todas estas nuevas clases es similar para cualquier estructura de datos. Esto permite que la ampliación de la herramienta con nuevas estructuras de datos no sea muy costosa.

Aunque en esta sección se ha abordado la ampliación de la aplicación con las colas de prioridad, la forma de actuar para añadir cualquier otra estructura nueva, sería similar a la descrita.

8.- Trabajo Futuro

A lo largo de toda la memoria, se ha intentado trasmitir el trabajo realizado para que la ampliación de la herramienta con nuevas estructuras de datos y sea lo más sencilla posible.

El departamento de Sistemas Informáticas ha pedido y le ha sido concedida una beca de innovación educativa. Esta beca va a permitir a Ana Isabel Saiz Jiménez, continuar trabajando en **VEDYA**, para su mejora y ampliación.

Durante el desarrollo de la beca se pretende añadir nuevas estructuras a la aplicación, tales como los árboles AVL, árboles rojinegros, colas de prioridad, tablas ordenadas y tablas Hash, ya que al menos se quiere conservar todas las estructuras de datos disponibles en versiones anteriores de **VEDYA**. Además, se quiere integrar en la nueva versión, los esquemas algorítmicos implementados el año 2004-2005, queriendo añadir algoritmos probabilistas.

9.- VALORACIÓN DEL TRABAJO REALIZADO

El desarrollo de este proyecto nos ha permitido profundizar en ciertos aspectos del funcionamiento del lenguaje de programación JAVA, adquiriendo una serie de destrezas que hemos utilizado para su realización. Entre estas destrezas adquiridas destacamos el manejo de hilos o threads, un conocimiento más completo sobre el método `paint()` de los componentes gráficos de swing y facilidad de utilización de herencia y poliformismo. Además, también hemos conseguido una comprensión completa del funcionamiento de las estructuras de datos que hemos implementado.

Se ha modificado **VEDYA** para que fuese una aplicación fácil e intuitiva de utilizar, intentando ofrecer al usuario una gran cantidad de opciones con las que pueda trabajar.

A lo largo del año, se ha realizado un gran esfuerzo para conseguir que las animaciones de las estructuras de datos fuesen originales, que llamasen la atención y que, sobre todo, sirvieran para comprender el funcionamiento de las estructuras de datos.

Creemos, que en líneas generales, hemos conseguido este objetivo, aunque nos hubiera gustado haber podido completar la aplicación, con al menos, todas las estructuras de datos que tenía la versión anterior de **VEDYA**.

A finales de curso, se realizó una presentación de la aplicación en cada una de las clases de Estructuras de Datos y de la Información, EDI, donde los alumnos pudieron ver por primera vez la nueva versión de **VEDYA**. Tuvo una buena acogida, e incluso hubo caras de grata sorpresa. Podemos decir que fue una experiencia muy positiva.

10.- GLOSARIO

- **Acción:** *unidad atómica e indivisible en las que se puede dividir una operación. Las operaciones que requieren animaciones simples, están compuestas por una única acción. Las operaciones que requieren animaciones complejas, están compuestas por varias acciones.*
- **Animación:** *es la ejecución visual de una operación.*
- **Árbol:** *es una estructura de datos formada por una colección de nodos, que puede estar vacía o no. Si no está vacía, el árbol estará formado por un nodo raíz, y cero o más subárboles, que están unidos a la raíz por otras tantas aristas. La aplicación no contendrá esta estructura de datos.*
- **Árbol binario:** *es una particularización de la estructura de datos Árbol. La característica que define a este tipo de Árboles, es que cada nodo puede tener como máximo dos hijos (subárboles binarios), llamados hijo izquierdo e hijo derecho.*
- **Árbol binario de búsqueda:** *es un caso concreto de la estructura de datos Árbol Binario. La característica que define a este tipo de Árboles Binarios, es que los valores de los nodos están ordenados. Este orden se define de la siguiente forma: el valor contenido en todos los nodos internos es mayor o igual que los valores contenidos en su hijo izquierdo o en cualquiera de los descendientes de ese hijo, y menor o igual que los valores contenidos en su hijo derecho o en cualquiera de los descendientes de ese hijo.*
- **Cola:** *es una estructura de datos lineal homogénea (en ella se pueden almacenar elementos de cualquier tipo, pero todos los elementos deben ser de ese mismo tipo), en la que los datos entran por un extremo y salen por el otro. La cola es una estructura FIFO (first-in, first-out) ya que el primer elemento de la cola será el primero en salir de ella.*
- **Cuadro de diálogo:** *es la etiqueta situada entre la barra de herramientas y el panel dibujo, donde se muestran comentarios que explican el funcionamiento de la aplicación.*
- **Cuadro de operaciones:** *es el panel que contiene todos los botones que realizan las operaciones sobre una estructura de datos. Está contenido dentro de la ventana, en la parte izquierda de la pantalla.*

- **EDGráfica:** nos referimos con este término, a cada una de las clases gráficas implementadas. Estas clases son: PilaGraficaUsuario, PilaGraficaEstatica, y PilaGraficaDinamica que heredan de PilaGrafica, ColaGraficaUsuario, ColaGraficaEstatica, ColaGraficaDinamica que heredan de ColaGrafica y ArbolBinarioBusquedaGraficoUsuario que hereda de ArbolBinarioBusquedaGrafico.
- **Elementos Asociados:** ventana que muestra información auxiliar asociada a una operación realizada sobre una estructura de datos.
- **Estructura de datos:** es una colección de datos cuya organización se caracteriza por las funciones definidas utilizadas para almacenar y acceder a elementos individuales de datos. Las estructuras de datos pueden descomponerse en los elementos que la forman. La manera en que se colocan los elementos dentro de la estructura afectará la forma en que se realicen los accesos a cada elemento. La colocación de los elementos y la manera en que se accede a ellos puede ser encapsulada.
- **Estados de una estructura:** son cada uno de las formas que adquiere una estructura de datos a la largo de una simulación. Gracias a los controles de simulación, podemos ir de un estado a otro de la estructura de datos con o sin animación.
- **Implementación:** para representar gráficamente las estructuras de datos, internamente se han implementado de una forma eficiente las propias estructuras de datos. Estas implementaciones son las que gobiernan las simulaciones de las estructuras..
- **Operación:** es cada una de las funciones (constructoras, modificadoras, observadoras) definidas para una estructura de datos.
- **Panel Animación:** panel en el que se muestran las estructuras de datos y sus animaciones. Está contenido dentro de la pestaña, en el centro de la pantalla.
- **PestañaED:** nos referimos con este término, a cada una de las clases pestañas implementadas para cada estructura de datos. Estas clases son: PestañaPila, PestañaCola y PestañaArbolBinarioBusqueda. Todas estas clases heredan de GeneradorPestaña.

- **Paso:** *cada una de las partes en las que se divide la ejecución de una acción. Generalmente se utilizan los pasos, para mostrar en los instantes correctos el comentario apropiado a la animación en ejecución.*
- **Pila:** *es una estructura de datos lineal y homogénea (en ella se pueden almacenar elementos de cualquier tipo, pero todos los elementos deben ser de ese mismo tipo). La pila es una estructura LIFO (last-in, first-out) ya que el último elemento insertado en la pila será el primero en salir de ella. El último elemento introducido en la pila se denomina cima.*
- **Registro operaciones:** *panel en el que se muestran todas las operaciones de una simulación. Está contenido dentro de la pestaña, en la parte derecha de la pantalla.*
- **Simulación:** *es la secuencia completa de operaciones definida sobre una estructura de datos. Se han creado unos controles que gobiernan las simulaciones, de forma que podemos movernos a lo largo de una simulación.*
- **Tramo:** *cada una de la partes en las que se dividen los movimientos de los elementos en una animación.*
- **Vector Elementos:** *vector que contiene los elementos gráficos utilizados para simular una estructura de datos.*
- **VentanaED:** *nos referimos con este término, a cada una de las clases ventanas implementadas para cada estructura de datos. Estas clases son: VentanaPila, VentanaCola y VentanaArbolBinarioBusqueda. Todas estas clases heredan de GeneradorVentana.*
- **VEDYA:** *nombre de la herramienta. Acrónimo de "Visualización de Estructuras de Datos y Algoritmos"*
- **Visualización:** *cada una de los distintos puntos de vista bajo los cuales se muestra una estructura de datos. Las visualizaciones pueden ser: usuario (visión de la estructura de datos bajo el punto de vista de su funcionamiento externo), implementación estática usuario (visión de la estructura de datos bajo el punto de vista de su implementación estática) e implementación dinámica usuario (visión de la estructura de datos bajo el punto de vista de su implementación dinámica).*

11.- BIBLIOGRAFÍA

- **Título**: Estructuras de datos y métodos algorítmicos: Ejercicios resueltos.
  Autores: Narciso Martí Oliet, Yolanda Ortega Mallén, José Alberto Verdejo López.
  Editorial: Pearson-Prentice-Hall, 2003. 1ª edición.
  Referencia en la memoria: [MOV03]

- **Título**: Diseño de programas. Formalismo y abstracción.
  Autor: Ricardo Peña Marí.
  Editorial: Prentice Hall, 2003, 2ªedición
  Referencia en la memoria: [Peñ03]

- **Título**: Fundamentos de algoritmia.
  Autores: G. Brassard, P. Bratley. Traducción de Rafael García-Bermejo.
  Revisión técnica: Narciso Martí, Ricardo Peña y Luís Joyanes Aguilar.
  Editorial: Prentice Hall, 2002. 1ª edición.
  Referencia en la memoria: [BB02]

- **Título**: Advanced Java 2 Platform: how to Program.
  Autores: H. M. Deitel, P. J. Deitel, S. E. Santry.
  Editorial: Prentice Hall, 2002. Colección: How to program series.
  Referencia en la memoria: [DDS02]

- **Título**: Data Structures and Algorithm Analysis in Java.
  Autor: Mark Allen Weiss,
  Editorial: Addison-Wesley, 1999.
  Referencia en la memoria: [Wei99]

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Página de Autorización

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Madrid, a 3 de Julio de 2007

Fdo. Ana Isabel Saiz Jiménez

Fdo. Pablo Soler Núñez

Fdo. Miguel Cayeiro García